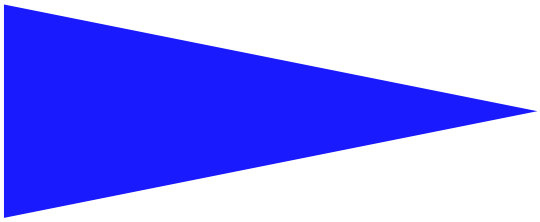
Computing stack maps with interfaces

Frédéric Besson, Thomas Jensen, Tiphaine Turpin

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

P U B L I C A T I O N
I N T E R N E
N° 1879 - version 2



COMPUTING STACK MAPS WITH INTERFACES

FRÉDÉRIC BESSON, THOMAS JENSEN, TIPHAINE TURPIN

ISSN 1166-8687

I R I S A
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Computing stack maps with interfaces

Frédéric Besson[*] , Thomas Jensen[**] , Tiphaine Turpin[***]

Systèmes symboliques
Projet Lande

Publication interne n ° 1879 — version 2[****]
initial version December 20, 2007 — revised version July 15, 2008 — 39 pages

Abstract: Lightweight bytecode verification uses stack maps to annotate Java bytecode programs with type information so that the checker only has to validate this typing, without having to do any data flow analysis. This report describes an improved analysis technique together with algorithms for optimizing the stack maps generated by the analyser. The improved analyser is based on a modified version of the abstract domain. This domain is simplified in its treatment of base values, keeping only the necessary information to ensure the memory safety property. It is richer in its representation of interface types, using the known Dedekind-MacNeille completion technique to construct abstract domain elements representing sets of interfaces. Tracking interface information allows to remove the dynamic checks at interface method invocations. We prove the memory safety property guaranteed by the verifier using an operational semantics whose distinguishing feature is a low-level memory model operating on untagged 32-bit values, as opposed to the standard, higher-level memory models using tagged base values. For bytecode that is typable without sets of types (this includes any code compiled from Java) we show how to prune the fix-point to obtain a stack map that can be validated without the interface set computations arising from this extension. In the context of lightweight verification, this is an advantage as it does not make the checking more complex or costly. The size of the stack maps is not significantly modified. Experiments show that the pruning can be done by reasonably efficient (though in theory exponential) algorithms that uses heuristics to explore the space of valid program typings from the least fixpoint generated by the analyser. Stack maps for three substantial test suites were correctly handled by the optimized (but incomplete) pruning algorithm.

Key-words: Bytecode verification, memory-safety, pruning, abstract interpretation

(Résumé : tsvp)

[*] IRISA/INRIA Rennes - Bretagne Atlantique
[**] IRISA/CNRS
[***] IRISA/Université de Rennes 1
[****] revised after publication in ECOOP'08

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Calcul de certificats avec interfaces

Résumé : La vérification de bytecode "lightweight" utilise des certificats pour anotter les programmes en bytecode Java avec une information de type, de telle sorte que le verifieur de bytecode n'ait plus qu'à vérifier ce typage, sans avoir besoin de faire une quelconque analyse de flot de données. Nous décrivons une technique de vérification améliorée, couplée à des algorithmes qui optimisent les certificats générés par l'analyseur. L'analyseur amélioré est fondé sur une version modifiée du domaine abstrait pour la vérification de bytecode. Ce domaine est simplifié en ce qui concerne le traitement des valeurs de base, et conserve seulement l'information nécessaire pour assurer la propriété de sûreté mémoire. Il est plus riche dans sa représentation des types interfaces, utilisant la technique connue de complétion de Dedekind-MacNeille pour construire des éléments du domaine abstrait qui représentent des ensembles d'interfaces. La vérification des interfaces permet de supprimer les contrôles dynamiques lors des appels de méthodes d'interfaces. Nous prouvons que la propriété de sûreté mémoire garantie par le vérifieur est assurée, en utilisant une sémantique opérationnelle qui se distingue par un modèle de la mémoire bas niveau opérant sur des valeurs de 32 bits non annotées, contrairement aux modèles mémoire standard, de plus haut niveau, qui utilisent des objects mémoires annotés. Pour du bytecode pouvant être typé sans recourir à des ensembles de types (ce qui inclut tout bytecode compilé à partir de Java), nous montrons comment élaguer le point-fixe pour obtenir un certificat qui peut être validé sans les calculs sur des ensembles d'interfaces qui surviennent à cause de cette extension. Dans le contexte de la vérification "lightweight", c'est un avantage puisque cela ne rend pas la validation plus compliquée ou coûteuse. La taille des certificat n'est pas modifiée de façon significative. L'expérience montre que l'élagage peut être effectué par des algorithmes raisonnablement efficaces (bien qu'exponentiels en théorie) qui utilisent des heuristiques pour explorer l'espace des typages valides d'un programme en partant du plus petit point-fixe généré par l'analyseur. Les certificats pour trois suites de tests conséquentes ont tous été correctement traités par l'algorithme d'élagage optimisé (mais incomplet).

Mots clés : Vérification de bytecode, sûreté mémoire, élagage, interprétation abstraite

Contents

1 Introduction

The Java bytecode verifier, which is part of the Java Virtual Machine [LY99] is a central component in Java security. It contains the static part of the safety checks that are performed on bytecode to ensure that it is "safe", in the sense that its execution conforms to a set of rules that entail that the program can only interact with its environment through the well-defined interface of invoking methods and reading or modifying objects. Together with the access control that is performed dynamically with these actions, and with the security policy that underlies the available API, this makes possible to execute untrusted bytecode without risk, for example web applets or mobile phone midlets.

While the standard bytecode verifier performs a dataflow analysis on the bytecode by fix-point iteration, the lightweight bytecode verifier [BLTY03] only checks a fix-point (which is called a *stack map*) that is shipped with the bytecode. It was originally designed for resource-constrained devices but the mainstream Java 2 Standard Edition (J2SE) is now moving to this kind of verification, and Java 6 uses a lightweight bytecode verifier with slightly enhanced stack maps (see JSR 202 [JSR06]). This is an instance of the more general Proof Carrying Code [Nec97] paradigm.

A particular issue for the type inference performed by in bytecode verification is the possibility for a class to implement several interfaces. The problem arises as soon as the language has multiple inheritance (only for interfaces, in the case of Java). This implies that the type hierarchy is not a lattice and prevents the computation of a unique most precise type for some variables, unless using sets of types. For simplicity, the choice made in the original verifiers (both standard and lightweight) was to ignore interfaces in bytecode verification and to make the necessary checks dynamically. This choice has been maintained in JSR 202.

We propose to extend the bytecode analysis to check interfaces statically, using conjunctions of types, and then to prune the result in order to obtain a stack map without conjunctions that can be fed to an almost unmodified checker. This technique does not work for every possible Java bytecode, but it applies to any bytecode obtained by compilation of Java. The result is that the dynamic checks on interface methods may be safely removed for free. Additionally, the pruned stack maps are sparser, and therefore they may be smaller. We describe the case of (idealized) Java bytecode, but the solution is not tied to this particular case, and would apply to other object oriented languages, even with general multiple inheritance.

In this report, the term analysis refers to the typing process that produces stack maps, checking is the validation of those stack maps on the consumer's side, and verification encompasses analysing, possibly pruning, and checking.

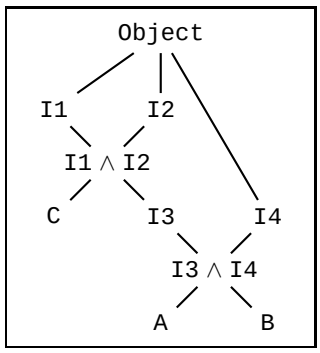
1.1 Motivating example

Figure 1 provides a small example which illustrates the existing verification and its extension to conjunctive types. Figure 1a represents a bytecode program written in pseudo Java, without type information. We suppose a type hierarchy with three classes `A`, `B` and `C` and four interfaces `I`i ($i \in [1, \ldots, 4]$) where `C` implements and `I3` extends `I1` and `I2`, and `A` and `B` implements `I3` and `I4`. Each interface `I`i declares a method `m`i. Figure 1b shows the completion of the type hierarchy that is used by our enhanced analyser, which adds the elements `I1` $\wedge$ `I2` and `I3` $\wedge$ `I4` to the type hierarchy.

The standard bytecode verifier ignores interfaces. Thus, in method `foo`, the variable `i` at program point 3 is given as type the first common super-class of `A` and `B`, *i.e.*, `Object`. Note also that the call to each method `m`i is in fact a call to the method of interface `I`i, where

```
void foo(boolean b) {
   if (b) {
(1)  i:=new A();
   } else {
(2)  i:=new B();
(3) }
(4) i.m1();
(5) i.m2();
}
```

(a) Main method



(b) Type hierarchy

Figure 1: A Java bytecode program and its type hierarchy

it is declared. When analysing those calls, the bytecode verifier only checks statically that the variable `i` contains an object type. At run-time, the JVM dynamically checks whether the object referenced by `i` implements `I1` and `I2`, before doing a lookup with respect to the dynamic type of `i`. If it is not the case, a run-time exception is thrown.

Our extended analizer will type the example program using conjunctions of types, and in particular, the variable `i` at program point 3 will have type `I3` $\wedge$ `I4`, which is propagated at program points 4 and 5. As this is a sub-type of both `I1` and `I2`, this ensures that the two method calls are safe. However, for the purpose of lightweight bytecode verification, it is desirable to avoid annotating the variables with conjunctions. The backtracking pruning algorithm proposed in Section 5 detects that in the above conjunction, only `I3` is needed to type the subsequent method invocations, hence it removes `I4`. In the resulting stack map, the variable `i` has therefore the type `I3` at program points 3, 4 and 5.

The example also shows that opting for a backward program analysis does not simplify the problem. An analyser which starts from the invocation sites and propagates these "uses" of a variable to the point of definition would still require the use of conjunctions and lead to a back-tracking algorithm. With such a technique, the variable `i` at program point 5 would get the type `I2`. The problem arises when typing `i` at program point 4: it must be the intersection of `I1` and `I2`, which requires either to introduce the conjunction `I1` $\wedge$ `I2`, or to choose one of the types `C` and `I3`. The right choice can only be made knowing the creation sites 1 and 2, hence the need for a backtraking algorithm.

1.2 Organisation of the report

We define the intraprocedural part of a small-step operational semantics with big-step calls for a subset of the Java bytecode (Section 2), with a low-level treatment of values that allows for a convincing definition of the memory safety property (which is the base for all other security properties). The analysis is presented in Section 3 in terms of an abstract interpretation [CC77], and proved correct. We define the notions of stack map and lightweight verification for a method, and state the main soundness lemma for a method. The interprocedural layer of the semantics is discussed in Section 4, where the soundness theorem for a complete program is proved. Section 5 describes the pruning algorithms. We give an efficient semi-algorithm that works in all but some well-identified, pathological cases, that do not seem to occur in average Java programs. We report on some experiments on

verifying Eclipse and a suite of Java MIDP midlets for mobile phones with a more complete prototype implementation in Section 6. Related work is discussed in Section 7 and Section 8 concludes.

1.3 Notations

Sets have long *italic* names, other constants and constants functions are in roman, with the exception of bytecode instructions for which we use sans serif. Meta-variables have short lowercase *italic* names, except that we write $C, I, F^\sharp$ for classes, interfaces and abstract transfer functions, respectively.

For sets a and b, we write $a \rightharpoonup b$ for the set of partial functions from a to b. We use the notation $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ to represent partial functions in extension. If f is a partial function, $\mathrm{dom}(f)$ is the domain of f, and any boolean expression e containing a sub-expression $f(e')$ implicitely means: $e' \in \mathrm{dom}(f) \wedge e$. Anonymous functions are denoted by lambda-abstractions or using a dot notation: for example, the expression $e[\cdot, \ldots, \cdot]$ stands for $\lambda x_1, \ldots, x_n.e[x_1, \ldots, x_n]$. We note $|x|$ for the cardinal of the set x, or the cardinal of its domain if x is a function. If f is a function (or a partial function), we note $f[x \leftarrow v]$ the function that maps x to v and any $y \neq x$ to $f(y)$.

Cartesian product takes precedence over other set operations: $\times \prec \cup, \rightarrow, \rightharpoonup$.

2 Intraprocedural semantics and memory safety

In this section we define formally our bytecode language and its semantics, and state the memory safety property that we ensure. We first define the semantics (and safety) of one single method, parameterised by the semantics of the (direct) method calls it may involve. We add the interprocedural part of the semantics (and the corresponding proof) in Section 4.

2.1 The Java bytecode language

We present a minimal subset of the language, abstracting away irrelevant features (such as the operand stack) while keeping the main aspects (objects, interface methods) that are relevant to typing. The subset is sufficiently representative for the results to extend to the whole Java bytecode.

2.1.1 Simplifications

The following features are absent from our language.

Operand stack The local operand stack is an aspect of the language (and of the verification process) independent of the abstract domain used for the verification so we omit it, replacing actual bytecode instructions by a three-addresses style instruction set that operates directly on local variables.

Other aspects Constant fields and static or interface members, exceptions, void methods, basic types other than int, sub-routines, threads, class and objects initialisation, access control (visibility), as well as as explicit type checks (the important checkcast instruction) are not considered. We discuss their impact on this study in Sections 6.1.1 and 6.1.2. Also, for conciseness we use less specialised instructions than the real virtual machine (for example, we merge iaload and aaload, ireturn and areturn).

2.1.2 Object oriented structure

The notations defined in this and following sections are summarized in Figure 12 in Appendix A. Let *ident* be the set of fully qualified Java identifiers.

Classes and interfaces We assume a set $class \subset ident$ of class names with a distinguished element Object, and a disjoint set $interface \subset ident$ of interface names. We define the set *type* of types recursively as:

$$type ::= \text{int} \quad | \quad C \quad | \quad I \quad | \quad t[]$$

where $C \in class$, $I \in interface$, and $t \in type$.

Type hierarchy The class hierarchy is modeled by the following three functions:

$$\begin{array}{lcrcl} \text{super} & : & class \setminus \{\text{Object}\} & \rightarrow & class \\ \text{implements} & : & class & \rightarrow & \mathcal{P}(interface) \\ \text{extends} & : & interface & \rightarrow & \mathcal{P}(interface) \end{array}$$

The function super must be the ancestor function of a tree with root Object:

$$\forall C \in class \quad \exists i \geq 0 \quad \text{super}^i(C) = \text{Object}.$$

Methods A method signature is made of an object type, an identifier and a list of parameter types. We assume a subset *msig* of such signatures which represents the methods that are declared in the program being verified:

$$msig \subset \{t.m(t_1, \ldots, t_n) \mid t \in type \setminus \{\text{int}\}, m \in ident, t_i \in type\}$$

Note that we have a virtual method signature if t is a class or an array type, and an interface method signature otherwise. In the following, we do not distinguish between the two, as we do not need to precisely model the lookup procedure. We let

$$\text{arity}(t.m(t_1, \ldots, t_n)) = n.$$

Each method signature has a return type given by the function

$$\text{result} : msig \rightarrow type.$$

Fields A field signature is made of a class name, an identifier and a type. We assume a subset *fsig* of such signatures:

$$fsig \subset \{C.f : t \mid C \in class, f \in ident, t \in type\}$$

Note that $C.f : t$ represents a field declared in class C, and consequently, the set of fields that are relevant for a given class of objects must be looked for in its super-classes too (see the function fields in Section 2.2.2).

2.1.3 The current method

As stated in the introduction, we first consider one execution of one "current" method, hence method-related definitions are not indexed by a signature.

Attributes of the current method We write *var* for the set of local variables of the method to verify and $arg \subseteq var$ for its set of formal parameters, whose types are described by the function

$$\mathrm{t}_{\mathrm{arg}} : arg \to type.$$

The return type of the current method is denoted by

$$\mathrm{t}_{\mathrm{ret}} \in type.$$

Program points are represented by the interval

$$ppoint = [0, |ppoint| - 1].$$

Instructions The instruction set is parameterised by *class*, *type*, *msig*, *fsig*, *var* and *ppoint*. We define the set *expr* of expressions and the set *instr* of instructions as follows. Here C ranges over classes, t over types, *ms* over method signatures, *fs* over field signatures, x, y, z, x_i over local variables, and p over program points.

$$\begin{array}{lcll} expr & ::= & n & n \in [-2^{31}, 2^{31} - 1] \\ & | & \text{null} \;\; | \;\; y + z \;\; | \;\; \mathsf{new}\; C \;\; | \;\; y.fs \;\; | \;\; \mathsf{new}\; t[y] \;\; | \;\; y[z] & \\ & | & y.ms(x_1, \ldots, x_{\mathrm{arity}(ms)}) & \\ \\ instr & ::= & x := e & e \in expr \\ & | & x.fs := y \;\; | \;\; x[y] := z \;\; | \;\; \mathsf{goto}\; p \;\; | \;\; \mathsf{if}\; x < y \;\; p \;\; | \;\; \mathsf{return}\; x & \end{array}$$

The method code is represented by the function

$$\mathrm{code} : ppoint \to instr$$

mapping program points to instructions. The last instruction $\mathrm{code}(|ppoint| - 1)$ must be either $\mathsf{goto}\; p$ for some p or $\mathsf{return}\; x$ for some x.

These last definitions enforce some well-formedness constraints on the code: the execution remains within the bounds of the code and cannot fall through the end, only valid local variables are referred to, and methods are always called with the right number of arguments. These properties are normally checked by the bytecode verifier prior to the type verifications.

2.2 Semantics

The operational semantics is defined as a small-step transition relation between program states (except for method calls which are big-step).

2.2.1 Discussion

First we discuss some design choices about the level and the kind of abstractions that are used.

Abstracting values and memory We choose to use a single data type of 32-bit values both for signed integer values and memory locations (note that objects and arrays are still annotated with their dynamic type, as in actual JVM implementations). This differs notably from most other formalisations where a disjoint set of locations is used (or equivalently, values are tagged with their type), this choice being only informally justified, as for example in [Pus99]:

> *"[...] the type information is not used to determine the operational semantics of (correct) JVM code."*

Barthe, Dufay, Jakubiec and de Sousa [BDJdS02] formalized this intuition by considering the actual virtual machine (which is called offensive) as an abstraction of the tagged (defensive) machine, and proving that the former correctly abstracts the latter, whenever the latter does not raise a type error (which is true for verifiable bytecode). Working directly with an untagged semantics immediately frees from of the risk of making unwanted implicit typing assumptions.

A precise model of the representation of objects and arrays in the memory is not necessary however. It is enough to use functions, state explicitly their domain and not use them out of it: any concrete representation, for example that maps these domains to sets of offsets, will conform to this model, provided the memory allocator keeps track of the range of objects and does not make them overlap.

Errors We make an important distinction between two kinds of errors:

- Runtime errors that are checked for dynamically and cause the virtual machine to raise an exception, such as accessing an array out of bounds or putting an element of the wrong type in it, are represented by the absence of transition.
- Actual type errors (called linking errors in the JVM specification) that violate the assumptions that a virtual machine implementation is allowed to make about the code (see [LY99]), such as dereferencing an integer, or accessing a non-existing field of an object, are represented by a transition to the special state error. This second kind of errors must be correctly handled by the bytecode verification, as the behavior of the virtual machine is unspecified for those cases, and in practice this can result in a crash (in the optimistic case) or the bypassing of access controls.

In the current JVM, the `invokeinterface` instruction should raise the exception `IncompatibleClassChangeError` if the receiver of the method does not implement the interface. Because our enhanced bytecode verifier will also type-check interfaces, we shift this exception from the class of runtime errors to the class of type errors. In our semantics, interface calls are dealt with like virtual calls and it is a type error if the receiver of an interface call does not implement the desired interface. Remark that the runtime errors raised in the explicit cast instruction (which we don't consider) are not removed by this technique.

2.2.2 Objects, arrays and states

The notations defined in this and following sections are summarized in Figure 13 in Appendix A. We write *word* for the set of 32-bit values. Values are used to represent signed integers as well as memory locations.

Objects We let $\text{fields} : \mathit{class} \to \mathcal{P}(\mathit{fsig})$ be the function that returns the set of (transitively inherited) fields of a class:

$$\text{fields}(C) = \{C.f : t \ \in \mathit{fsig}\} \cup \begin{cases} \text{fields}(\text{super}(C)) & \text{if } C \neq \text{Object} \\ \emptyset & \text{otherwise} \end{cases}$$

An object is a pair $\langle C, o\rangle$ where $C \in \mathit{class}$ and $o : \text{fields}(C) \to \mathit{word}$ gives the value of the relevant fields. We write *object* for the set of objects.

$$\mathit{object} = \{\langle C, o\rangle \mid C \in \mathit{class}, o : \text{fields}(C) \to \mathit{word}\}$$

Arrays We let *array* be the set of arrays, annotated with their element type (which can be an array type):

$$array = \{\langle t, a\rangle \mid t \in type, a : [0, n-1] \to word, n \geq 0\}$$

Program states We define *heap* as the sets of partial mappings from non-zero values to objects and arrays:

$$heap = word \setminus \{0\} \rightharpoonup (object \cup array).$$

The memory allocator is represented by a partial function

$$\text{alloc} : heap \rightharpoonup word \setminus \{0\}$$

that maps a heap h to a value that is not defined in h (the absence of value represent the failure of the allocation)[1]:

$$\forall h \in heap \quad \text{alloc}(h) = v \implies v \notin \text{dom}(h).$$

A program state $s = \langle h, l, p\rangle$ consists of a heap, a (total) mapping from variables to values, and a program point:

$$state = heap \times (var \to word) \times ppoint$$

2.2.3 Dynamic typing

We first recall the standard sub-typing order $\preceq \subseteq type \times type$ induced by the functions super, implements and extends. Note that, in J2SE, every array type is a sub-type of the two interfaces Cloneable and Serializable (which therefore we assume exist).

$$\frac{}{t \preceq t} \qquad \frac{t \preceq t' \quad t' \preceq t''}{t \preceq t''} \qquad \frac{t \preceq t'}{t[] \preceq t'[]}$$

$$\frac{}{C \preceq \text{super}(C)} \qquad \frac{I \in \text{implements}(C)}{C \preceq I} \qquad \frac{I' \in \text{extends}(I)}{I \preceq I'}$$

$$\frac{}{I \preceq \text{Object}} \qquad \frac{}{t[] \preceq \text{Cloneable}} \qquad \frac{}{t[] \preceq \text{Serializable}}$$

The precise definition of this order is not important, the results generalize to slightly different settings and do not strongly rely on the last three rules in particular. The key properties that are actually used in the following are:

- that $\preceq$ is a partial order
- the existence of the maximum element (which is called Object in this case)
- the covariant ordering of array types (third rule in the first line)

and of course the link with the functions super, implements and extends, for the language that we consider.

The dynamic typing relation $h \vdash v : t$ between heaps, 32-bit values and types is defined as follows:

$$\frac{}{h \vdash v : \text{int}} \qquad \frac{}{h \vdash 0 : t}$$

$$\frac{h(v) = \langle C, o\rangle \in object \quad C \preceq t}{h \vdash v : t} \qquad \frac{h(v) = \langle t, a\rangle \in array \quad t[] \preceq t'}{h \vdash v : t'}$$

[1] This is not completely accurate, as the allocation obviously depends on the needed size.

It must be noted that this relation can be implemented efficiently by traversing the type hierarchy above the value whose type is being checked upward. This is crucial since it is used by the concrete semantics of array assignement.

2.2.4 Method calls

Let *bigstep* be the type of big-step semantics for methods.

$$\mathit{bigstep} = \mathcal{P}\left(\bigcup_{n\geq 0}((\mathit{heap}\times\mathit{word}\times\mathit{word}^n)\times(\mathit{heap}\times\mathit{word}\cup\{\text{error}\}))\right)$$

Let $bs \in \mathit{bigstep}$ and $\langle\langle h, \mathit{this}, \mathit{args}\rangle, r\rangle \in bs$. *this* represents the object on which the method is to be invoked, and *args* represents the list of the arguments. The result r is either the error constant or a pair $\langle h', v\rangle \in \mathit{heap}\times\mathit{word}$ where v is the returned value and h' is the heap obtained by running the method from the initial heap h. For now, the direct method calls that may arise during the execution of the current method are represented by associating a big step transition relation

$$\xrightarrow{ms} \in \mathit{bigstep}$$

to each method signature *ms* (note that the relation for one method signature may correspond to several actual methods due to dynamic binding). As the relation is supposed to represent every possible call without any assumption on the arguments, is must be defined even for ill-typed ones, possibly with the result error. Also, we make no hypothesis on the correctness of the invoked method yet, thus the error state may be returned even for arguments of the right type. The absence of transition from a particular list of arguments represents non-termination or a runtime exception.

2.2.5 Transition relation

The semantics is given by the transition relation

$$\rightarrow \subseteq \mathit{state}\times(\mathit{state}\cup\mathit{heap}\times\mathit{word}\cup\{\text{error}\})$$

defined in Figure 2. Some intermediate definitions make use of the additional constant $\perp$ to denote stuck computations.

Comments on some rules A couple of features in this semantics merit explanation.

- Writing to a field (see Figure 2b) always succeeds (provided the field exists for the target object), even if the value that is written is not of the right type. This is not a safety violation by itself, only a future misuse of this bad value (for example, accessing a field it doesn't have) would be an error.
- Writing to an array always triggers a dynamic check[2] and the execution is naturally stuck in case that the value stored in the array is not a sub-type of the array's own type, or if the index is out of bounds (an exception is raised in the real virtual machine).
- In Figure 2c, it is important to remember that a method call can occur with ill-typed arguments, and that the invoked method itself can be ill-typed, hence the rule for the error state.

[2]This is unavoidable with the covariant arrays of the Java type system.

$$
\begin{array}{lcl}
[\![n]\!]_{h,l} & = & n \quad \text{(32-bit signed encoding)} \\
[\![\text{null}]\!]_{h,l} & = & 0 \\
[\![y+z]\!]_{h,l} & = & l(y)+l(z) \\
[\![y.fs]\!]_{h,l} & = & \left\{\begin{array}{ll} o(fs) & \text{if } h(l(y)) = \langle C, o\rangle \in \mathit{object} \ \wedge \ \mathit{fs} \in \text{fields}(C) \\ \bot & \text{if } l(y) = 0 \\ \text{error} & \text{if} \quad l(y) \neq 0 \ \wedge \ l(y) \notin \text{dom}(h) \\ & \quad \vee \ h(l(y)) \notin \mathit{object} \\ & \quad \vee \ h(l(y)) = \langle C, o\rangle \in \mathit{object} \ \wedge \ \mathit{fs} \notin \text{fields}(C) \end{array}\right. \\
[\![y[z]]\!]_{h,l} & = & \left\{\begin{array}{ll} a(l(z)) & \text{if } h(l(y)) = \langle t, a\rangle \in \mathit{array} \ \wedge \ 0 \leq l(z) < |a| \\ \bot & \text{if } l(y) = 0 \ \vee \ h(l(y)) = \langle t, a\rangle \in \mathit{array} \ \wedge \neg\, 0 \leq l(z) < |a| \\ \text{error} & \text{if } l(y) \neq 0 \ \wedge \ l(y) \notin \text{dom}(h) \ \vee \ h(l(y)) \notin \mathit{array} \end{array}\right.
\end{array}
$$

(a) Semantics $[\![e]\!]_{h,l} \in \mathit{word} \cup \{\text{error}, \bot\}$ of a side-effect free expression e in context h, l

$$
\begin{array}{lcl}
[\![x := e]\!](h,l) & = & \left\{\begin{array}{ll} h, l[x \leftarrow [\![e]\!]_{h,l}] & \\ \qquad \text{if } [\![e]\!]_{h,l} \notin \{\bot, \text{error}\} & \\ \bot \quad \text{if } [\![e]\!]_{h,l} = \bot & \\ \text{error} \quad \text{if } [\![e]\!]_{h,l} = \text{error} & \end{array}\right\} \quad \begin{array}{l} \text{if} \quad e \notin \{y.ms(\cdots), \\ \qquad \mathsf{new}\ C, \\ \qquad \mathsf{new}\ t[y]\} \end{array} \\
[\![x := \mathsf{new}\ C]\!](h,l) & = & \left\{\begin{array}{ll} h[v \leftarrow \langle C, \lambda \mathit{fs} \in \text{fields}(C).0\rangle], l[x \leftarrow v] & \\ \qquad \text{if alloc}(h) = v & \\ \bot \quad \text{if } h \notin \text{dom(alloc)} & \end{array}\right. \\
[\![x := \mathsf{new}\ t[y]]\!](h,l) & = & \left\{\begin{array}{ll} h[v \leftarrow \langle t, \lambda i \in [0, l(y)].0\rangle], l[x \leftarrow v] & \\ \qquad \text{if } l(y) \geq 0 \ \wedge \ \mathit{alloc}(h) = v & \\ \bot \quad \text{if } l(y) < 0 \ \vee \ h \notin \text{dom(alloc)} & \end{array}\right. \\
[\![x.fs := y]\!](h,l) & = & \left\{\begin{array}{ll} h[l(x) \leftarrow \langle C, o[fs \leftarrow l(y)]\rangle], l & \\ \qquad \text{if } h(l(x)) = \langle C, o\rangle \in \mathit{object} \ \wedge \ \mathit{fs} \in \text{fields}(C) & \\ \bot \quad \text{if } l(x) = 0 & \\ \text{error} \quad \text{if} \quad l(x) \neq 0 \ \wedge \ l(x) \notin \text{dom}(h) & \\ \qquad \vee \ h(l(x)) \notin \mathit{object} & \\ \qquad \vee \ h(l(x)) = \langle C, o\rangle \in \mathit{object} \ \wedge \ \mathit{fs} \notin \text{fields}(C) & \end{array}\right. \\
[\![x[y] := z]\!](h,l) & = & \left\{\begin{array}{ll} h[l(x) \leftarrow \langle t, a[l(y) \leftarrow l(z)]\rangle], l & \\ \qquad \text{if} \quad h(l(x)) = \langle t, a\rangle \in \mathit{array} & \\ \qquad \wedge\, h \vdash l(z) : t \ \wedge \ 0 \leq l(y) < |a| & \\ \bot \quad \text{if} \quad l(x) = 0 & \\ \qquad \vee \quad l(x) = \langle t, a\rangle \in \mathit{array} & \\ \qquad \wedge \ \neg\, (h \vdash l(z) : t \ \wedge \ 0 \leq l(y) < |a|) & \\ \text{error} \quad \text{if} \quad l(x) \neq 0 \ \wedge \ l(x) \notin \text{dom}(h) & \\ \qquad \vee \ h(l(x)) \notin \mathit{array} & \end{array}\right.
\end{array}
$$

(b) Semantics $[\![i]\!] : \mathit{heap} \times (\mathit{var} \rightarrow \mathit{word}) \rightarrow (\mathit{heap} \times (\mathit{var} \rightarrow \mathit{word}) \cup \{\text{error}, \bot\})$ of a non-branching intraprocedural instruction i

$$
\frac{i \notin \{\mathsf{goto}\ p, \mathsf{if} \cdots, \mathsf{return}\ x, x := y.ms(\cdots)\} \quad [\![i]\!](h,l) \neq \bot}{h, l \xrightarrow{i} [\![i]\!](h,l)} \qquad \frac{}{h, l \xrightarrow{\mathsf{return}\ x} h, l(x)}
$$

$$
\frac{h, l(y), l(x_1), \ldots, l(x_n) \xrightarrow{ms} h', v}{h, l \xrightarrow{x := y.ms(x_1, \ldots, x_n)} h', l[x \leftarrow v]} \qquad \frac{h, l(y), l(x_1), \ldots, l(x_n) \xrightarrow{ms} \text{error}}{h, l \xrightarrow{x := y.ms(x_1, \ldots, x_n)} \text{error}}
$$

(c) Semantics $\xrightarrow{i} \subseteq (\mathit{heap} \times (\mathit{var} \rightarrow \mathit{word})) \times (\mathit{heap} \times (\mathit{var} \rightarrow \mathit{word}) \cup \mathit{heap} \times \mathit{word} \cup \{\text{error}\})$ of a non-branching instruction i

Figure 2: Semantics of Java bytecode

$$\frac{\text{code}(p) \notin \{\mathsf{goto}\ p', \mathsf{if}\cdots\} \quad h, l \xrightarrow{\text{code}(p)} h', l'}{\langle h, l, p\rangle \to \langle h', l', p+1\rangle} \qquad \frac{\text{code}(p) \notin \{\mathsf{goto}\ p', \mathsf{if}\cdots\} \quad h, l \xrightarrow{\text{code}(p)} h', v}{\langle h, l, p\rangle \to h', v}$$

$$\frac{\text{code}(p) \notin \{\mathsf{goto}\ p', \mathsf{if}\cdots\} \quad h, l \xrightarrow{\text{code}(p)} \text{error}}{\langle h, l, p\rangle \to \text{error}} \qquad \frac{\text{code}(p) = \mathsf{goto}\ p'}{\langle h, l, p\rangle \to \langle h, l, p'\rangle}$$

$$\frac{\text{code}(p) = \mathsf{if}\ x < y\ \ p' \quad l(x) < l(y)}{\langle h, l, p\rangle \to \langle h, l, p'\rangle} \qquad \frac{\text{code}(p) = \mathsf{if}\ x < y\ \ p' \quad l(x) \geq l(y)}{\langle h, l, p\rangle \to \langle h, l, p+1\rangle}$$

(d) Small step transition relation $\to \subseteq \mathit{state} \times (\mathit{state} \cup \mathit{heap} \times \mathit{word} \cup \{\text{error}\})$

Figure 2: Semantics of Java bytecode (continued)

2.3 Memory safety

We give a modular definition of memory safety that is stronger than what is actually needed for a complete program: it includes the preservation of the well-typedness of the heap, and the fact that the heap is only extended. This property requires some prior definitions to express accurate invariants about the heap. See Theorem 1 in Section 4 for a minimal statement.

Well typed heaps The following relation expresses that a heap is consistent.

$$\frac{\forall v \in \mathit{word} \setminus \{0\} \quad \left\{\begin{array}{ll} & v \notin \text{dom}(h) \\ \vee & h(v) = \langle C, o\rangle \in \mathit{object} \\ & \wedge\ \forall C'.f : t \in \text{fields}(C) \quad h \vdash o(C'.f : t) : t \\ \vee & h(v) = \langle t, a\rangle \in \mathit{array} \\ & \wedge\ \forall i \in [0, |a| - 1] \quad h \vdash a(i) : t \end{array}\right.}{h \vdash h}$$

Ordering on heaps The relation $\Subset$ expresses the preservation of existing objects between two heaps.

$$\frac{\forall v \in \mathit{word} \setminus \{0\} \quad \left\{\begin{array}{l} v \notin \text{dom}(h) \\ \vee\ h(v) = \langle C, o\rangle \in \mathit{object}\ \wedge h'(v) = \langle C, o'\rangle \in \mathit{object} \\ \vee\ h(v) = \langle t, a\rangle \in \mathit{array}\ \wedge h'(v) = \langle t, a'\rangle \in \mathit{array} \end{array}\right.}{h \Subset h'}$$

Modular memory-safety The following definition introduces the general safety property for the transition relation associated with a method. We need to give two variants of it since we use slightly different formalisations for the transition relation of the current method and the relations representing method calls. As errors are immediately propagated in the semantics, it is sound to define the safety as the unreachability of error in the outermost invocation.

Definition 1 *The relation* $\to \subseteq \mathit{state} \times (\mathit{state} \cup (\mathit{heap} \times \mathit{word}) \cup \{\text{error}\})$ *is safe with respect to* t_{arg} *and* t_{ret} *if for all* h, l *such that* $h \vdash h \ \wedge\ \forall x \in \mathit{arg} \quad h \vdash l(x) : \text{t}_{\text{arg}}(x)$ *then* $\langle h, l, 0\rangle \not\to^* \text{error}$ *and* $\langle h, l, 0\rangle \to^* h', v \implies h \Subset h' \wedge h' \vdash h' \wedge h' \vdash v : \text{t}_{\text{ret}}$.

Similarly, a transition relation $\xrightarrow{ms} \in \mathit{bigstep}$ *is safe with respect to the signature* $ms = t.m(t_1, \ldots, t_n)$ *if for all* $h, v, v_1, \ldots, v_n$ *such that* $h \vdash h \ \wedge\ h \vdash v : t \ \wedge\ \forall i \leq n \quad h \vdash v_i : t_i$ *then* $h, v, v_1, \ldots, v_n \not\xrightarrow{ms} \text{error}$ *and* $h, v, v_1, \ldots, v_n \xrightarrow{ms} h', v' \implies h \Subset h' \wedge h' \vdash h' \wedge h' \vdash v' : \text{result}(ms)$.

Memory safety and security In this study we choose to focus on memory safety which is just one aspect of the security of Java bytecode. It basically unsures that the virtual machine will not crash when executing the program. However, the safe execution of untrusted bytecode typically requires stronger properties, like the informal idea that a program should not be able to forge a pointer to a heap location that it is not supposed to have access to. It is not easy to define formally this requirement without instrumenting the semantics. For example with the semantics that we just defined we could prove that the heap location returned by any method whose return type is an reference type be either unnallocated in the initial heap or reachable from the (reference) arguments on which the method was invoked. This is ensured by the analysis of Section 3 without any modification: only the proofs have to be extended by strengthening the concretisation function for the abstract domain. But this definition is still naive, as it does not distinguish between private or public fields, nor does it account for the fact that an untrusted program can be given controlled access to some (private) objects through the invocation of trusted methods from the API. Nevertheless, even thoug the memory-safety itself does not imply any access restriction property, the analysis by which we ensure memory safety represents a large part of what is needed to ensure security, as shown by Leroy and Rouaix [LR98] who formalizes such stronger security properties, and give relatively local sufficient conditions, in addition to well-typedness, for an applet to be safe with respect to these properties.

3 Extended bytecode typing

In this section we present an extended abstract domain for interface-aware bytecode verification and prove it sound with respect to the semantics. The main difference with the standard bytecode verification is the use of interfaces in types, which make the runtime check in the "invokeinterface" instruction unnecessary. Another difference is that integers are not distinguished from the $\top_v$ value. This simplifies the presentation and also the stack maps. The verification is a modular process where each individual method is checked with respect to its given signature, assuming that the calls that it triggers (possibly of the same method) are correctly described by these signatures. The interprocedural aspects will be treated in Section 4. The definitions of this section are summarized in Figure 14 in Appendix A.

3.1 Abstract domain

The abstract domain elements are called *stack maps* in the context of Java bytecode verification, as they normally map program points to abstract operand stacks. We keep this name even though the stack is absent in our setting.

3.1.1 Stack maps

Our abstract domain associates a type to each variable at each program point. This type is either $\top_v$ (for non-reference values), null, or a conjunction of object (or array) types.

$$
\begin{array}{lcll}
\mathit{value}^\sharp & ::= & \text{null} \mid \top_v & \\
& \mid & t_1 \wedge \cdots \wedge t_n & n \geq 1, t_i \in \mathit{type} \setminus \{\text{int}\}, \\
& & & \forall i,j \leq n \quad t_i \preceq t_j \implies i = j \\
\mathit{state}^\sharp & = & \mathit{var} \to \mathit{value}^\sharp & \\
\mathit{map} & = & \mathit{ppoint} \to \mathit{state}^\sharp &
\end{array}
$$

$$
\mathit{state}^{\sharp\,\top_s}_{\bot_s} = \mathit{state}^\sharp \cup \{\bot_s, \top_s\}
$$

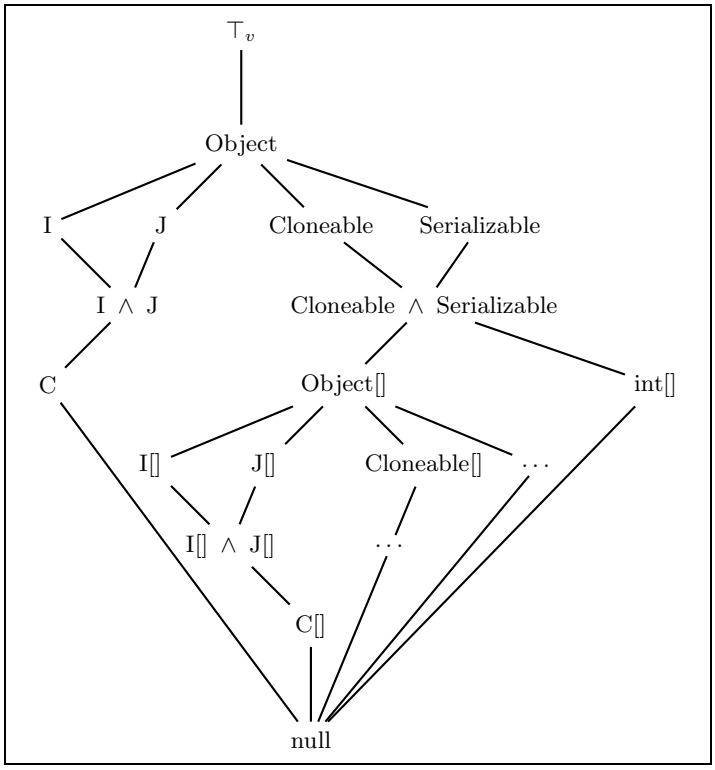


Figure 3: Abstract domain for a hierarchy containing the interfaces I, J and the class C. The conjunctions whose concretisation is empty have been omitted for conciseness.

The two special abstract states $\bot_s$ and $\top_s$ indicate respectively an unreachable program point and the possibility of the (concrete) error state being reachable. We also define the abbreviation $state^\sharp{}_{\bot_s}^{\top_s}$. Conjunctions are defined up to the order, *i.e.*, $t \wedge t' = t' \wedge t$.

A conjunction is to be interpreted as the set of objects that are a member of every atomic type in it. Note that we only consider conjunctions of unordered atomic types. This is necessary to be able to define an order on abstract values and not just a pre-order, and also to make the concretisation (almost) injective (as adding a super-type of another conjunct does not change the concretisation of a conjunction). Another isomorphic solution is to consider upward-closed (with respect to $\preceq$) sets of atomic types. We choose the first representation which is more compact and hence allows us to compute least upper bounds efficiently (see Section 3.1.4). An example abstract domain $value^\sharp$ is presented in Figure 3.1.

The following definition identifies a subset of stack maps in which we want to choose a "certificate" to send with the current method.

Definition 2 *A stack map $m \in map$ is conjunction-free if all of its conjunctions $t_1 \wedge \cdots \wedge t_n$ are reduced to one element (*i.e.*, $n = 1$).*

3.1.2 Concretisation

We define the concretisation functions $\gamma_h : \mathit{value}^\sharp \to \mathcal{P}(\mathit{word})$, $h \in \mathit{heap}$, $\gamma_p : \mathit{state}^{\sharp\,\top_s}_{\bot_s} \to \mathcal{P}(\mathit{state} \cup \{\text{error}\})$, $p \in \mathit{ppoint}$ and $\gamma : (\mathit{value}^\sharp \cup \{\top_s\}) \to \mathcal{P}((\mathit{heap} \times \mathit{word}) \cup \{\text{error}\})$ by

$$\begin{array}{lcl}
\gamma_h(\top_v) & = & \mathit{word} \\
\gamma_h(\text{null}) & = & \{0\} \\
\gamma_h(t_1 \wedge \cdots \wedge t_n) & = & \{v \in \mathit{word} \mid \forall i \leq n \quad h \vdash v : t_i\}
\end{array}$$

$$\begin{array}{lcl}
\gamma_p(\bot_s) & = & \emptyset \\
\gamma_p(\top_s) & = & \mathit{state} \cup \{\text{error}\} \\
\gamma_p(l^\sharp) & = & \{\langle h, l, p\rangle \in \mathit{state} \mid h \vdash h \ \wedge\ \forall x \in \mathit{var} \quad l(x) \in \gamma_h(l^\sharp(x))\}
\end{array}$$

$$\begin{array}{lcl}
\gamma(\top_s) & = & \mathit{heap} \times \mathit{word} \cup \{\text{error}\} \\
\gamma(v^\sharp) & = & \left\{h, v \mid h \vdash h \ \wedge\ v \in \gamma_h(v^\sharp)\right\}.
\end{array}$$

3.1.3 Partial order

The partial orders $\sqsubseteq_v \subseteq \mathit{value}^\sharp \times \mathit{value}^\sharp$ and $\sqsubseteq \subseteq \mathit{state}^{\sharp\,\top_s}_{\bot_s} \times \mathit{state}^{\sharp\,\top_s}_{\bot_s}$ are defined by the following rules:

$$\frac{}{\text{null} \sqsubseteq_v v^\sharp} \qquad \frac{}{v^\sharp \sqsubseteq_v \top_v} \qquad \frac{\forall j \leq n' \quad \exists i \leq n \quad t_i \preceq t'_j}{t_1 \wedge \cdots \wedge t_n \ \sqsubseteq_v \ t'_1 \wedge \cdots \wedge t'_{n'}}$$

$$\frac{}{\bot_s \sqsubseteq s^\sharp} \qquad \frac{}{s^\sharp \sqsubseteq \top_s} \qquad \frac{\forall x \in \mathit{var} \quad l^\sharp(x) \sqsubseteq l'^\sharp(x)}{l^\sharp \sqsubseteq l'^\sharp}$$

The fact that those relations are partial orders follows from $\preceq$ being a partial order and from the restriction to conjunctions of unordered atomic types.

3.1.4 Least upper bound

The (commutative) least upper bound operators $\sqcup_v : \mathit{value}^\sharp \times \mathit{value}^\sharp \to \mathit{value}^\sharp$ and $\sqcup : \mathit{state}^{\sharp\,\top_s}_{\bot_s} \times \mathit{state}^{\sharp\,\top_s}_{\bot_s} \to \mathit{state}^{\sharp\,\top_s}_{\bot_s}$ are defined by

$$\begin{array}{lcl}
\text{null} \sqcup_v v^\sharp & = & v^\sharp \\
v^\sharp \sqcup_v \top_v & = & \top_v \\
t_1 \wedge \cdots \wedge t_n \ \sqcup_v \ t'_1 \wedge \cdots \wedge t'_{n'} & = & \left\{\begin{array}{l}
\text{let } \mathcal{T} = \\
\quad \{t \in \mathit{type} \mid \exists i \leq n, j \leq n' \quad t_i \preceq t \ \wedge\ t'_j \preceq t\} \\
\text{in} \\
\quad \bigwedge\{t \in \mathcal{T} \mid \forall t' \in \mathcal{T} \quad t' \preceq t \implies t' = t\}
\end{array}\right. \\
\bot_s \sqcup s^\sharp & = & s^\sharp \\
s^\sharp \sqcup \top_s & = & \top_s \\
l^\sharp \sqcup l'^\sharp & = & \lambda x. l^\sharp(x) \sqcup_v l'^\sharp(x).
\end{array}$$

The least upper bound of two (non-empty) conjunctions is always defined (and non-empty), because Object is a super type of all reference types (including interfaces and array types). The second line in the least upper bounds of two conjunctions ensures that we keep only maximal atoms.

The actual computation of the least upper bound of two conjunctions can be performed efficiently: as only minimal types $t \in \mathcal{T}$ will be kept, it is sufficient to find the first super-class and/or super-interfaces of each pair t_i, t'_j, which is done by traversing the hierarchy above t_i and t'_j (array types pose no problem either).

3.1.5 Transfer function

Figure 4 defines the abstract semantics as two relations

$$\longrightarrow \subseteq \mathit{ppoint} \times (\mathit{state}^\sharp \to \mathit{state}^{\sharp\,\top_s}_{\bot_s}) \times \mathit{ppoint}$$

and

$$\longrightarrow \subseteq \mathit{ppoint} \times (\mathit{state}^\sharp \to \mathit{value}^\sharp).$$

3.2 Correctness of the abstraction

We now prove the consistency of the previous definitions.

3.2.1 Partial order

The following lemma ensures the consistency of the partial order with respect to the concretisation, which is crucial for the correctness of the verification.

Lemma 1 *For all $h \in \mathit{heap}$ and $v^\sharp, v'^\sharp \in \mathit{value}^\sharp$, if $v^\sharp \sqsubseteq_v v'^\sharp$ then $\gamma_h(v^\sharp) \subseteq \gamma_h(v'^\sharp)$. For all $p \in \mathit{ppoint}$ and $s^\sharp, s'^\sharp \in \mathit{state}^{\sharp\,\top_s}_{\bot_s}$, if $s^\sharp \sqsubseteq s'^\sharp$ then $\gamma_p(s^\sharp) \subseteq \gamma_p(s'^\sharp)$. For all $v^\sharp, v'^\sharp \in \mathit{value}^\sharp \cup \{\top_s\}$, if $v'^\sharp = \top_s \vee v^\sharp \sqsubseteq_v v'^\sharp$ then $\gamma(v^\sharp) \subseteq \gamma(v'^\sharp)$.*

The proof follows from the definitions.

3.2.2 Least upper bound

The least upper bound operator is used during the fix-point computation and must be correct for the generated stack map to be accepted by the checker.

Lemma 2 *For all $v^\sharp, v'^\sharp \in \mathit{value}^\sharp$, $v^\sharp \sqcup_v v'^\sharp \sqsupseteq_v v^\sharp$ and $v^\sharp \sqcup_v v'^\sharp \sqsupseteq_v v'^\sharp$. For all $s^\sharp, s'^\sharp \in \mathit{state}^{\sharp\,\top_s}_{\bot_s}$, $s^\sharp \sqcup s'^\sharp \sqsupseteq s^\sharp$ and $s^\sharp \sqcup s'^\sharp \sqsupseteq s'^\sharp$.*

The proof follows from the definitions.

3.2.3 Transfer function

The core of the correctness of the analysis resides in the following lemma which says that the concrete transition relation is correctly approximated by the abstract one.

Lemma 3 *Suppose that for every signature ms the relation $\xrightarrow{ms}$ is safe with respect to ms (see Definition 1). Let $s \in \mathit{state}$, $r \in \mathit{state} \cup (\mathit{heap} \times \mathit{word}) \cup \{\text{error}\}$, $p \in \mathit{ppoint}$ and $l^\sharp \in \mathit{state}^\sharp$ such that $s \to r$ and $s \in \gamma_p(l^\sharp)$. Then one of the following holds:*

1. *$p \xrightarrow{F^\sharp} p'$ and $r \in \gamma_{p'}(F^\sharp(l^\sharp))$ for some p' and $F^\sharp$, or*
2. *$p \xrightarrow{F^\sharp}$ and $r \in \gamma(F^\sharp(l^\sharp))$ for some $F^\sharp$.*

$$
\begin{array}{lcl}
[\![n]\!]^\sharp_{l^\sharp} & = & \top_v \\
[\![\text{null}]\!]^\sharp_{l^\sharp} & = & \text{null} \\
[\![y + z]\!]^\sharp_{l^\sharp} & = & \top_v \\
[\![\mathsf{new}\ C]\!]^\sharp_{l^\sharp} & = & C \\
[\![\mathsf{new}\ t[y]]\!]^\sharp_{l^\sharp} & = & t[] \\
[\![y.fs]\!]^\sharp_{l^\sharp} & = & \begin{cases} t & \text{if } fs = C.f : t \ \wedge\ l^\sharp(y) \sqsubseteq_v C \\ \top_s & \text{otherwise} \end{cases} \\
[\![y[z]]\!]^\sharp_{l^\sharp} & = & \begin{cases} t & \text{if } l^\sharp(y) = t[] \quad t \in \mathit{type} \\ \bot_s & \text{if } l^\sharp(y) = \text{null} \\ \top_s & \text{if } l^\sharp(y) \notin \{\text{null}\} \cup \{t[] \mid t \in \mathit{type}\} \end{cases} \\
[\![y.ms(x_1, \ldots, x_n)]\!]^\sharp_{l^\sharp} & = & \begin{cases} \text{result}(ms) & \\ \quad \text{if} \quad ms = t.m(t_1, \ldots, t_n) & \\ \quad\quad \wedge\ l^\sharp(y) \sqsubseteq_v t \ \wedge\ \forall i \leq n \quad l^\sharp(x_i) \sqsubseteq_v t_i & \\ \top_s \quad \text{otherwise} & \end{cases}
\end{array}
$$

(a) Abstract semantics $[\![e]\!]^\sharp_{l^\sharp} \in \mathit{value}^\sharp \cup \{\bot_s, \top_s\}$ of an expression e in the abstract context $l^\sharp$

$$
\begin{array}{lcl}
[\![x := e]\!]^\sharp(l^\sharp) & = & \begin{cases} l^\sharp[x \leftarrow [\![e]\!]^\sharp_{l^\sharp}] & \\ \quad \text{if } [\![e]\!]^\sharp_{l^\sharp} \notin \{\bot_s, \top_s\} & \\ \bot_s \quad \text{if } [\![e]\!]^\sharp_{l^\sharp} = \bot_s & \\ \top_s \quad \text{if } [\![e]\!]^\sharp_{l^\sharp} = \top_s & \end{cases} \\
[\![x.fs := y]\!]^\sharp(l^\sharp) & = & \begin{cases} l^\sharp & \text{if} \quad fs = C.f : t \\ & \quad \wedge\ l^\sharp(x) \sqsubseteq_v C \ \wedge\ l^\sharp(y) \sqsubseteq_v t \\ \top_s & \text{otherwise} \end{cases} \\
[\![x[y] := z]\!]^\sharp(l^\sharp) & = & \begin{cases} l^\sharp & \text{if } l^\sharp(x) = t[] \quad t \in \mathit{type} \\ \bot_s & \text{if } l^\sharp(x) = \text{null} \\ \top_s & \text{if } l^\sharp(y) \notin \{\text{null}\} \cup \{t[] \mid t \in \mathit{type}\} \end{cases}
\end{array}
$$

(b) Abstract semantics $[\![i]\!]^\sharp : \mathit{state}^\sharp \to \mathit{state}^{\sharp\,\top_s}_{\bot_s}$ of a non-branching instruction i ($i \neq \mathsf{return}\ x$)

$$
\frac{\text{code}(p) \notin \{\mathsf{return}\ x, \mathsf{goto}\ p', \mathsf{if} \cdots\}}{p \overset{[\![\text{code}(p)]\!]^\sharp}{\longrightarrow} p+1} \qquad \frac{\text{code}(p) = \mathsf{goto}\ p'}{p \overset{\lambda l^\sharp.\, l^\sharp}{\longrightarrow} p'}
$$

$$
\frac{\text{code}(p) = \mathsf{if}\ x < y\ \ p'}{p \overset{\lambda l^\sharp.\, l^\sharp}{\longrightarrow} p'} \qquad \frac{\text{code}(p) = \mathsf{if}\ x < y\ \ p'}{p \overset{\lambda l^\sharp.\, l^\sharp}{\longrightarrow} p+1}
$$

$$
\frac{\text{code}(p) = \mathsf{return}\ x}{p \overset{\lambda l^\sharp.\, l^\sharp(x)}{\longrightarrow}}
$$

(c) Abstract transition relations $\longrightarrow \subseteq \mathit{ppoint} \times (\mathit{state}^\sharp \to \mathit{state}^{\sharp\,\top_s}_{\bot_s}) \times \mathit{ppoint}$ and $\longrightarrow \subseteq \mathit{ppoint} \times (\mathit{state}^\sharp \to \mathit{value}^\sharp)$

Figure 4: Abstract semantics of Java bytecode

Furthermore, the functions $F^\sharp$ that label the abstract transition relations are monotone.

Proof: By definition we have that $s = \langle h, l, p\rangle$. We reason by case on the instruction code(p).

- Case goto p': then $r = \langle h, l, p'\rangle$ and $p \xrightarrow{\lambda l^\sharp.l^\sharp} p'$. From $\langle h, l, p\rangle \in \gamma_p(l^\sharp)$ we derive $\langle h, l, p'\rangle \in \gamma_{p'}(l^\sharp)$ and we are in case 1.
- Case if $x < y$ p' is similar.
- Case return x: then $r = h, l(x)$ and $p \xrightarrow{\lambda l^\sharp.l^\sharp(x)}$ which falls in case 2 because $l(x) \in \gamma(l^\sharp(x))$.
- Case code(p) $\notin$ {goto p', if $\cdots$, return x}: then $p \xrightarrow{[\![\text{code}(p)]\!]^\sharp} p+1$ and thus we have to prove that $r \in \gamma_{p+1}([\![\text{code}(p)]\!]^\sharp(\langle h, l, p\rangle))$.
  - Case $x := y.ms(x_1, \ldots, x_n)$: then we have $ms = t.m(t_1, \ldots, t_n)$. If $l^\sharp(y) \sqsubseteq_v t$ and $\forall i \leq n \quad l^\sharp(x_i) \sqsubseteq_v t_i$ then $h \vdash l(y) : t$ and $\forall i \leq n \quad h \vdash l(x_i) : t_i$ (by definition of γ_h). This implies (by safety hypothesis on $\xrightarrow{ms}$) that $r = \langle h', l[x \leftarrow v], p+1\rangle$ with $h \Subset h'$, $h' \vdash h'$ and $h' \vdash v : \text{result}(ms)$. From $l \in \gamma_h(l^\sharp), h \Subset h'$ and $h' \vdash v : \text{result}(ms)$ we deduce that $l[x \leftarrow v] \in \gamma_{h'}(l^\sharp[x \leftarrow \text{result}(ms)])$ and we conclude since $[\![x := y.ms(x_1, \ldots, x_n)]\!]^\sharp(l^\sharp) = l^\sharp[x \leftarrow \text{result}(ms)]$. Otherwise, $[\![x := y.ms(x_1, \ldots, x_n)]\!]^\sharp(l^\sharp) = \top_s$ and this is trivial.
  - Case code(p) $\notin$ {goto p', if $\cdots$, return $x, x := y.ms(\cdots)$}: then we know that $[\![\text{code}(p)]\!](h, l) \neq \bot$ and furthermore, we have either $r = \text{error} \wedge [\![\text{code}(p)]\!](h, l) = \text{error}$ or $r = \langle h', l', p+1\rangle \wedge h', l' = [\![code(p)]\!](h, l)$. Therefore we must prove that either $[\![\text{code}(p)]\!](h, l) = \text{error} \wedge [\![\text{code}(p)]\!]^\sharp(l^\sharp) = \top_s$ or $[\![\text{code}(p)]\!](h, l) = h', l' \wedge \langle h', l', p+1\rangle \in \gamma_{p+1}([\![\text{code}(p)]\!]^\sharp(l^\sharp))$.
    - Case $x :=$ new C: then $[\![x :=$ new $C]\!](h, l) = h[v \leftarrow o], l[x \leftarrow v]$ for some v and o such that $h \vdash o : C$ and $v \notin \text{dom}(h)$. Therefore, we deduce that $h[v \leftarrow o] \vdash h[v \leftarrow o]$, $h[v \leftarrow o] \vdash l[x \leftarrow v](y) : l^\sharp(y)$ for $y \neq x$ and $h[v \leftarrow o] \vdash l[x \leftarrow v](x) : C$. As $[\![x :=$ new $C]\!]^\sharp(l^\sharp) = l^\sharp[x \leftarrow C]$, we can conclude.
    - Case $x :=$ new $t[y]$ is similar.
    - Case $x := e$, $e \notin \{y.ms(\cdots), \text{new } C, \text{new } t[y]\}$: then we know that $[\![e]\!]_{h,l} \neq \bot$, and we have either $[\![x := e]\!](h, l) = \text{error} \wedge [\![e]\!]_{h,l} = \text{error}$ or $[\![x := e]\!](h, l) = h, l[x \leftarrow [\![e]\!]_{h,l}]$. Symetrically, either $[\![x := e]\!]^\sharp(l^\sharp) = \top_s \wedge [\![e]\!]^\sharp_{l^\sharp} = \top_s$ or $[\![x := e]\!]^\sharp(l^\sharp) = l^\sharp[x \leftarrow [\![e]\!]^\sharp_{l^\sharp}]$. Therefore we must prove that either $[\![e]\!]_{h,l} = \text{error} \wedge [\![e]\!]^\sharp_{l^\sharp} = \top_s$ or $[\![e]\!]_{h,l} \in \gamma_h([\![e]\!]^\sharp_{l^\sharp})$.
      - Case $e \in \{n, y+z\}$: this is trivial since $[\![e]\!]_{h,l} \neq \text{error}$ and $[\![e]\!]^\sharp_{l^\sharp} = \top_v$
      - Case null follows from $[\![e]\!]_{h,l} = 0$ and $[\![e]\!]^\sharp_{l^\sharp} = \text{null}$.
      - Case $y.fs$: from $[\![\text{code}(p)]\!](h, l) \neq \bot$ we get $l(y) \neq 0$. Let $fs = C.f : t$. If $l^\sharp(y) \sqsubseteq_v C$ then $h(l(y)) = \langle C', o\rangle \in O$ and $C' \preceq C$ (by definition of γ_h), which implies $fs \in \text{fields}(C')$ and then $[\![y.fs]\!]_{h,l} = o(fs)$. As $h \vdash h$, it follows that $h \vdash o(fs) : t$. On the other hand, we know that $[\![y.fs]\!]^\sharp_{l^\sharp} = t$ and we conclude. otherwise $[\![y.fs]\!]^\sharp_{l^\sharp} = \top_s$ and we are done.
      - Case $y[z]$ is similar.

∗ Case $x.fs := y$: from $[\![\text{code}(p)]\!](h,l) \neq \bot$ we get $l(x) \neq 0$. Let $fs = C.f : t$. If $l^\sharp(x) \sqsubseteq_v C$ and $l^\sharp(y) \sqsubseteq_v t$ then $h(l(x)) = \langle C', o\rangle \in O$ with $C' \preceq C$ and $h \vdash l(y) : t$ (by definition of γ_h), which implies that $fs \in \text{fields}(C')$ and then $[\![x.fs := y]\!](h,l) = h[l(x) \leftarrow \langle C', o[fs \leftarrow l(y)]\rangle], l$. As $h \vdash h$ and $h \vdash l(y) : t$, it follows that $h[l(x) \leftarrow \langle C', o[fs \leftarrow l(y)]\rangle] \vdash h[l(x) \leftarrow \langle C', o[fs \leftarrow l(y)]\rangle]$ and $\forall z \in var \quad l(z) \in \gamma_{h[l(x)\leftarrow\langle C', o[fs\leftarrow l(y)]\rangle]}(l^\sharp(z))$. On the other hand, we know that $[\![x.fs := y]\!]^\sharp(l^\sharp) = l^\sharp$ and we conclude. otherwise $[\![x.fs := y]\!]^\sharp(l^\sharp) = \top_s$ and we are done.

∗ Case $x[y] := z$ is similar.

The monotonicity of the functions $F^\sharp$ follows from the definitions. □

3.3 Analysis and checking

The following definition introduce the notion of witness of the current method whose signature is given by t_{arg} and t_{ret}.

Definition 3 *A witness is a stack map $m \in map$ such that*

1. $\forall x \in var \quad m(0)(x) \sqsupseteq_v \begin{cases} \text{t}_{\text{arg}}(x) & \text{if } x \in arg \\ \top_v & \text{otherwise} \end{cases}$
2. $\forall p, p' \in ppoint \quad p \xrightarrow{F^\sharp} p' \implies F^\sharp(m(p)) \sqsubseteq m(p')$
3. $\forall p \in ppoint \quad p \xrightarrow{F^\sharp} \implies F^\sharp(m(p)) \sqsubseteq_v \text{t}_{\text{ret}}$

Note that by definition of stack maps, witnesses contain no $\top_s$ or $\bot_s$. This correspond respectively to the assumptions that the code should type without error, and that even dead code should be typable. This second condition is necessary for the pruning to work.

The following lemma shows that the memory safety property can be ensured by simply checking that some given stack map is a witness, and shows how to compute the least witness. The next section will show that, for Java programs, the least witness can be pruned resulting in a witness without conjunction.

Lemma 4 *Suppose that every relation $\xrightarrow{ms}$ is safe with respect to ms (see Definition 1). If there exists a witness m then the relation $\rightarrow$ is safe with respect to t_{arg} and t_{ret}. Moreover, the least witness[3] (if there exists a witness) can be computed by fixpoint iteration.*

Proof: The first point follows directly from the previous definitions and lemmas: assume that $h \vdash h$ and $\forall x \in arg \quad h \vdash l(x) : \text{t}_{\text{arg}}(x)$. We prove by induction that if $\langle h, l, 0\rangle \rightarrow^* r$ then either $r = \langle h', l', p'\rangle \in \gamma_{p'}(m(p'))$ for some p' and $h \Subset h'$ or $r = h', v \in \gamma(\text{t}_{\text{ret}})$ and $h \Subset h'$.

- Let $x \in var$. If $x \in arg$ then from $h \vdash l(x) : \text{t}_{\text{arg}}(x)$ and $m(0)(x) \sqsupseteq_v \text{t}_{\text{arg}}(x)$ (condition 1 of Definition 3) we get that $l(x) \in \gamma_h(\text{t}_{\text{arg}}(x)) \sqsubseteq_v \gamma_h(m(0)(x))$ (Lemma 1). Otherwise, $m(0)(x) \sqsupseteq_v \top_v$ and $\gamma_h(\top_v) \ni l(x)$. In both cases, we have that $l(x) \in \gamma_h(m(0)(x))$. As $h \vdash h$, we obtain $\langle h, l, 0\rangle \in \gamma_0(m(0))$. And obviously, $h \Subset h$.
- Assume that $s = \langle h', l', p'\rangle \in \gamma_{p'}(m(p'))$ with $h \Subset h'$ and $s \rightarrow r$. Then, applying Lemma 3 we get two cases.

[3]The abstract state $\bot_s$ is absent from witnesses by definition, but the state $\lambda x \in var.\text{null}$ is a minimum of $state^\sharp$. Thus, if the set of witnesses is not empty, it has a least element.

– If $p' \xrightarrow{F^\sharp} p''$ and $r \in \gamma_{p''}(F^\sharp(m(p')))$ then by condition 2 of Definition 3 we have $F^\sharp(m(p')) \sqsubseteq m(p'')$ and we conclude that $r \in \gamma_{p''}(m(p''))$. As stack maps contain no $\top_s$, we conclude that $r = \langle h'', l'', p'' \rangle$. The fact that $h' \Subset h''$ follows from the definitions, as a step of the semantics never deletes, moves or modifies the type of an existing object. By transitivity, we get $h \Subset h''$.

– If $p' \xrightarrow{F^\sharp}$ and $r \in \gamma(F^\sharp(m(p')))$ then by condition 3 of Definition 3 we have $F^\sharp(m(p')) \sqsubseteq_v \mathrm{t_{ret}}$ and we deduce that $r \in \gamma(\mathrm{t_{ret}})$, and thus $r = h', v$ (with the same h' as in state s). So we have $h \Subset h'$ by hypothesis.

This proves that $\langle h, l, 0 \rangle \not\rightarrow^*$ error. Also, if $\langle h, l, 0 \rangle \rightarrow^* h, v$ we are in the second case: $h', v \in \gamma(\mathrm{t_{ret}})$ and $h \Subset h'$ (that is, $h' \vdash h' \wedge h' \vdash v : \mathrm{t_{ret}}$). This proves the first point. The second is a standard result of the abstract interpretation theory, knowing that the lattice $state^\sharp{}_{\perp_s}^{\top_s}$ satisfies the "finite ascending chain" condition and that the functions $F^\sharp$ in the abstract semantics are monotone. □

4 Interprocedural glue

We now formalise the whole semantics for the set of methods that compose a program and present the last induction step, showing that the safety of individual method entails, together with the consistency checks performed at load time (sub-typing of overridden methods, etc.), the safety of the whole program with respect to a big-step semantics for method calls. The following definitions and proofs are given only to justify the choice of analysing a method in an intraprocedural way, representing the (possibly recursive) methods calls by fixed relations, and are completely independant from the rest of the report.

4.1 Semantics

We first complete the definition of the concrete semantics to account for a set of method bodies, dynamic method lookup and recursive method calls. The new definitions introduced in this section are summarized in Figure 15 in Appendix A.

4.1.1 From the current method to a complete program

We assume that the set of parameters of the method considered so far is totally ordered: $arg = \{this, p_1, \ldots, p_n\}$ (in real bytecode this would be given implicitely by the signature, the variables names being natural integers, arguments first). The terminating executions of the current method are represented by a big-step relation

$$\underbrace{\dot{\rightarrow}, (var, arg, ppoint, \mathrm{code})}_{\longrightarrow} \in bigstep.$$

We make explicit its "parameters", *i.e.*, the function

$$\dot{\rightarrow} : msig \rightarrow bigstep$$

associating to each signature ms the relation $\overset{ms}{\rightarrow}$ that we used (Section 2.2.4) to represent direct calls to ms, and the body

$$(var, arg, ppoint, \mathrm{code})$$

of the method we considered. Relation $\underbrace{\overset{\cdot}{\rightarrow}, (var, arg, ppoint, \text{code})}_{\longrightarrow}$ is defined as follows:

$$\frac{l(\text{this}) = v \quad \forall i \leq n \quad l(p_i) = v_i \quad \langle h, l, 0\rangle \rightarrow^* r \quad r \notin state}{h, v, v_1, \ldots, v_n \underbrace{\overset{\cdot}{\rightarrow}, (var, arg, ppoint, \text{code})}_{\longrightarrow} r}$$

Remark that the two versions of Definition 1 are consistent with this definition: the small-step semantics $\rightarrow$ of the current method of name m is safe with respect to $\mathrm{t}_{\mathrm{arg}}$ and $\mathrm{t}_{\mathrm{ret}}$ if and only if its big-step semantics (that we just defined) is safe with respect to the signature

$$ms = \mathrm{t}_{\mathrm{arg}}(\text{this}).m(\mathrm{t}_{\mathrm{arg}}(p_1), \ldots, \mathrm{t}_{\mathrm{arg}}(p_n)),$$

assuming that

$$\text{result}(ms) = \mathrm{t}_{\mathrm{ret}}.$$

Now we generalise this notation, by considering a partial mapping

$$\text{body} : msig \rightharpoonup \{(var, arg, ppoint, \text{code}) \text{ as in Section 2.1.3}\},$$

which represents the whole program by giving an implementation to some method signatures (the function body is partial because interface methods have no implementation). Section 2 can thus be seen as the definition of the higher order function

$$\underbrace{\overset{\cdot,\cdot}{\longrightarrow}}_{} : (msig \rightarrow bigstep) \times \{\text{body}(ms) \mid ms \in msig\} \rightarrow bigstep.$$

Accordingly, Lemma 4 reads: if $\overset{\cdot}{\rightarrow}$ is such that every $\overset{ms}{\rightarrow}$ is safe with respect to ms and if there exists a witness m for a given method body $\text{body}(ms_0)$, then $\underbrace{\overset{\cdot}{\rightarrow}, \text{body}(ms_0)}_{\longrightarrow}$ is safe with respect to ms_0.

4.1.2 Dynamic method lookup

The details of the concrete lookup procedure are not needed to establish the correctness of the bytecode verification. We just represent it by a partial function

$$\text{lookup} : \{t, t'.m(t_1, \ldots, t_n) \in (type \setminus \{\text{int}\}) \times msig \mid t \preceq t'\} \rightharpoonup class$$

such that

$$\begin{aligned} &\text{lookup}(t, t'.m(t_1, \ldots, t_n)) = C \implies \\ &\quad t \preceq C \\ &\wedge\ C.m(t_1, \ldots, t_n) \in msig \\ &\wedge\ \text{result}(C.m(t_1, \ldots, t_n)) \preceq \text{result}(t'.m(t_1, \ldots, t_n)). \end{aligned}$$

This assumption correspond to the checks that are performed at class loading time, in addition to the hypothesis on the method lookup procedure itself.

We can now associate to each method signature a big-step semantics that correspond to the lookup of this signature on a particular object or array (not just to executing the code of this signature if any). The semantics

$$\underbrace{\overset{\cdot}{\rightarrow}, ms}_{\longrightarrow} \in bigstep$$

of a method signature ms is defined by the following two rules:

$$\frac{d > 0 \quad ms = t.m(t_1, \ldots, t_n) \quad \neg\, h \vdash v : t}{h, v, v_1, \ldots, v_n \overbrace{\xrightarrow{\cdot}, ms}^{\longrightarrow} \text{error}}$$

$$\frac{\begin{array}{c} v \neq 0 \\ h(v) = \langle C, o\rangle \in \mathit{object} \ \wedge \ t = C \\ \vee\ h(v) = \langle t', a\rangle \in \mathit{array} \ \wedge \ t = t'[] \\ ms = t''.m(t_1, \ldots, t_n) \\ t \preceq t'' \end{array} \qquad \begin{array}{c} \mathrm{lookup}(t, ms) = C' \\ ms' = C'.m(t_1, \ldots, t_n) \\ ms' \in \mathrm{dom}(\mathrm{body}) \\ h, v, v_1, \ldots, v_n \overbrace{\xrightarrow{\cdot}, \mathrm{body}(ms')}^{\longrightarrow} r \end{array}}{h, v, v_1, \ldots, v_n \overbrace{\xrightarrow{\cdot}, ms}^{\longrightarrow} r}$$

4.1.3 Recursive method invocation

Finally, the semantics $\xrightarrow{\cdot}$ of the program is defined as the least fix-point of the (continuous) function

$$\mathrm{S} : (\mathit{msig} \to \mathit{bigstep}) \to (\mathit{msig} \to \mathit{bigstep})$$

defined by

$$\mathrm{S}(\xrightarrow{\cdot})(ms) = \overbrace{\xrightarrow{\cdot}, ms}^{\longrightarrow}.$$

In other words, for all $ms \in \mathit{msig}$,

$$\xrightarrow{ms} = \bigcup_{d \geq 0} \left(\left(\mathrm{S}^d(\lambda ms \in \mathit{msig}.\emptyset) \right) (ms) \right)$$

Intuitively, d is the maximum depth of the method calls in this formula.

4.2 Correctness of the modular verification

Theorem 1 lifts the correctness lemma to the complete program, thus showing the soundness of the intraprocedural view.

Theorem 1 *If there exists a witness for every method body then for every method signature* $ms \in \mathit{msig}$, $\xrightarrow{ms}$ *is safe with respect to* ms. *In particular, for a method* main *of class* C,

$$\{1 \mapsto \langle \mathrm{C}, \lambda fs \in \mathrm{fields}(\mathrm{C}).0\rangle\}, 1 \overset{\mathrm{C.main}()}{\not\longrightarrow^*} \text{error}.$$

Proof: If $h, v, v_1, \ldots, v_n \xrightarrow{ms}_d r$ then $(h, v, v_1, \ldots, v_n), r \in \mathrm{S}^d(\lambda ms.\emptyset)$ for some $d \geq 0$. Therefore, as $\emptyset$ is safe with respect to any signature, it is enough to prove that if for some $\xrightarrow{\cdot} \in \mathit{msig} \to \mathit{bigstep}$, $\xrightarrow{ms}$ is safe with respect to ms for all $ms \in \mathit{msig}$, then this is also true for $\mathrm{S}(\xrightarrow{\cdot})$.

So, let $\xrightarrow{\cdot} \in \mathit{msig} \to \mathit{bigstep}$ such that for all ms, $\xrightarrow{ms}$ is safe with respect to ms. Let $ms_0 = t''.m(t_1, \ldots, t_n)$, and $h, v, v_1, \ldots, v_n, r$ such that $h \vdash h$, $h \vdash v : t''$, $\forall i \leq n \quad h \vdash v_i : t_i$ and $h, v, v_1, \ldots, v_n \overbrace{\xrightarrow{\cdot}, ms_0}^{\longrightarrow} r$. Then only the second rule applies, so we have $h \vdash v : t$, $t \preceq t''$, and

$h, v, v_1, \ldots, v_n \underbrace{\xrightarrow{\cdot}, \text{body}(ms')}_{\longrightarrow} r$ where $ms' = C'.m(t_1, \ldots, t_n)$ where $C' = \text{lookup}(t, ms_0)$. As $\xrightarrow{ms}$ is safe for every ms, and since there exists a witness for body(ms'), we can apply Lemma 4 and we get that $\underbrace{\xrightarrow{\cdot}, \text{body}(ms')}_{\longrightarrow}$ is safe with respect to ms'. By hypothesis on lookup, we know that $t'' \preceq C'$, which implies that $h \vdash v : C'$. Since $h \vdash h$ and $\forall i \leq n \quad h \vdash v_i : t_i$, we can deduce that $r \neq \text{error}$ and $r = h, v'$ with $h \Subset h'$, $h' \vdash h'$ and $h' \vdash v' : \text{result}(ms')$. Again by hypothesis on lookup, we have $\text{result}(ms') \preceq \text{result}(ms_0)$ and therefore $h' \vdash v' : \text{result}(ms_0)$. This proves that, $\underbrace{\xrightarrow{\cdot}, ms_0}_{\longrightarrow}$ is safe, and we obtain the desired property for S($\xrightarrow{\cdot}$). □

5 Lightweight verification by fix-point pruning

In the previous section we formalised a bytecode analysis extended to interfaces, using conjunctions of types in the abstract domain. The drawback of this extension is its computational cost, especially in terms of memory, that could make it unapplicable on the smallest Java capable devices. We will now present an additional step to the lightweight verification setting that removes the need for computations of sets of types on the consumer side by computing a witness without conjunction, if the safety of the program does not rely on them.

5.1 Stack map checking without conjunctions

We first present an algorithm that allows this extended verification to work without the overhead of a checker manipulating conjunctions of interfaces, when it is possible. Then we argue that this technique works for programs compiled from Java in particular.

5.1.1 Witnesses without conjunction

The key hypothesis of the pruning algorithm that we describe is the existence of a witness without conjunction. This is not the case of all programs, and Figure 5 shows a method that has no such witness (the program is shown in pseudo Java but one has to write it directly in bytecode). But this holds for most programs.

As explained in the previous section, the checking of bytecode mainly consist in the verification of the conditions of Definition 3, which reduces to computations of the functions $F^\sharp$ of the abstract semantics, and abstract ordering checks $\sqsubseteq$ between abstract states. As the $F^\sharp$s can never "create" a conjunction of types (this is easily verified on the definition), we can see that the checking of a conjunction-free witness can be performed without manipulating conjunctions.[4]

5.1.2 Fix-point pruning

The algorithm of Figure 6 optimistically searches for such a witness, until one is found or the search space is exhausted. It start from the least witness (if it exists)

$$\text{lfp} \in map$$

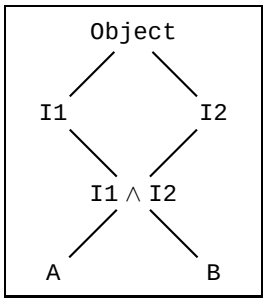
[4]In the real process, the value of the witness is only sent for some program points and the remaining values are reconstructed at checking time, but no least upper bound is involved, thus the property still holds.

```
interface I1 {void m1();}
interface I2 {void m2();}

class A implements I1, I2 {
    public void m1() {}
    public void m2() {}
}

class B implements I1, I2 {
    public void m1() {}
    public void m2() {}
}
```

(a) Class and interface declarations



(b) Type hierarchy

```
public void main(String  args) {
    I1 ∧ I2 i = args.length > 0 ? new A() : new B();
    i.m1();
    i.m2();
}
```

(c) Main method

Figure 5: A safe Java bytecode program that has no conjunction-free witness

let $w = \lambda p \in ppoint.\lambda x \in var.\top_v$ and $W = ppoint$
while $W \neq \emptyset$ do
  take $p \in W$ (and remove it)
  choose a maximal $l^\sharp \in state^\sharp$ such that
    $l^\sharp$ is without conjunction and $\mathrm{lfp}(p) \sqsubseteq l^\sharp \sqsubseteq w(p)$
    and $\forall p' \in ppoint \quad p \xrightarrow{F^\sharp} p' \implies$
      $F^\sharp(l^\sharp) \neq \top_s$
      $\wedge\ F^\sharp(l^\sharp) \sqsubseteq w(p')$
    $\wedge\ p \xrightarrow{F^\sharp} \implies F^\sharp(l^\sharp) \sqsubseteq_v \mathrm{t}_{\mathrm{ret}}$
  if $l^\sharp \neq w(p)$ then
    $w := w[p \leftarrow l^\sharp]$
    $W := W \cup \{p' \mid p' \xrightarrow{F^\sharp} p\}$
  end if
done
return w

Figure 6: Naive (complete) pruning algorithm

computed by direct analysis, and traverses the set of conjunction-free stack maps that are greater than lfp, until a witness is reached. We know that if there exists a witness without conjunction, it must belong to this set, by definition of the least fix-point. Fortunately, the finite ascending chain condition satisfied by the lattice of stack maps ensures that the search space is finite and therefore the process terminates (which is interesting in case there is no

such witness). Therefore this algorithm is complete in the sense that if a solution exists, it will find it.

More precisely, the idea is to start from $\top_v$ at each program point and each variable and replace those values by lesser ones until a witness is reached. The non-deterministic instruction "choose" is to be interpreted as follows: if the choice fails at any point (*i.e.*, there is no $v^\sharp$ satisfying the required conditions) then we backtrack to a previous choice. The algorithm can terminate either by returning a witness, or by returning nothing, if every combination of choices eventually gets stuck. As for the strategy used to implement the work-set (instruction "take"), we found that a stack without duplicates was an efficient and simple heuristic. Note that this second sort of choice is never undone and does not cause further backtracking.

Correctness The following theorem formalizes the fact that the algorithm of Figure 6 is sound and complete (when a witness without conjunction exists).

Theorem 2 *The complete pruning algorithm always terminates, either by returning a stack map w or by a failure in the choice of $l^\sharp$. If it terminates by returning some w, then w is a conjunction-free witness. Furthermore, if a conjunction-free witness w exists, the algorithm will return such a witness.*

Proof: It follows from the definition of $l^\sharp$ in the algorithm that w always decreases with respect to $\sqsubseteq$ during one branch of the execution (the proof is left to the reader). As the lattice satisfies the finite ascending chain condition, w is eventually stable, at which point W strictly decreases (with respect to $\subseteq$), which ensures the termination of every branch. Since the choice of $l^\sharp$ is only finitely branching, the global backtracking algorithm also terminates.

It is clear that the variable w always holds an abstract state without conjunction (because the value $l^\sharp$ that replaces $w(p)$ in the loop has no conjunction). Additionally, the following invariant is maintained during the execution of the loop:

$$\begin{array}{l} \forall p \in \mathit{ppoint} \setminus W \qquad \forall p' \in \mathit{ppoint} \quad p \xrightarrow{F^\sharp} p' \implies \begin{array}{r} F^\sharp(w(p)) \neq \top_s \\ \wedge\ F^\sharp(w(p)) \sqsubseteq w(p') \end{array} \\ \qquad\qquad \wedge\ p \xrightarrow{F^\sharp} \implies F^\sharp(w(p)) \sqsubseteq_v \mathrm{t}_{\mathrm{ret}} \end{array} .$$

The proof is by induction:

- Initially this is obvious since $W = \mathit{ppoint}$.
- Assume that this is true at some state denoted by w_{old}, W_{old}. Let p and $l^\sharp$ be the values considered in the loop, and w, W the next state. Let $p' \notin W$. Then either $p' \notin W_{old} \ \wedge\ p' \neq p$ or $p' = p$. In both cases it is clear that

$$\begin{array}{l} \forall p'' \in \mathit{ppoint} \quad p' \xrightarrow{F^\sharp} p'' \implies \begin{array}{r} F^\sharp(w(p')) \neq \top_s \\ \wedge\ F^\sharp(w(p')) \sqsubseteq w_{old}(p'') \end{array} \\ \wedge\ p' \xrightarrow{F^\sharp} \implies F^\sharp(w(p')) \sqsubseteq_v \mathrm{t}_{\mathrm{ret}} \end{array} .$$

  In the first case, this is because $w(p') = w_{old}(p')$ and because of the invariant at step w_{old}, W_{old} ; in the second case, because $w(p') = l^\sharp$ and by definition of $l^\sharp$. This property is almost what we want, except that we must replace $w_{old}(p'')$ by $w(p'')$. Thus, to prove the invariant at the next step, it is enough to show that $\forall p'' \in \mathit{ppoint} \quad p' \xrightarrow{F^\sharp} p'' \implies w_{old}(p'') = w(p'')$. We reason by case on p'': if $p'' = p$, then as $p' \notin W$, the definition of W implies that $l^\sharp = w_{old}(p)$, and since $w(p) = l^\sharp$ we get the result. If $p'' \neq p$, this is trivial.

This proves the soundness.

As for completeness, consider an execution where we always choose an $l^\sharp$ such that $l^\sharp \sqsupseteq w_0(p)$, where w_0 is the witness without conjunction whose existence we assume. We prove the following invariant: this choice is always possible (and in particular the algorithm never gets stuck) and the variable w of the algorithm satisfies $w \sqsupseteq w_0$. By induction:

- Initially this is obvious by definition of w.
- Assume that $w_{old} \sqsupseteq w_0$ at some state denoted by w_{old}, W_{old}. Let p and $l^\sharp$ be the values considered in the loop, and w, W the next state. Then by definition of w_0 and since $w_{old} \sqsupseteq w_0$ we have that $w_0(p)$ is without conjunction and $w_0(p) \sqsupseteq \mathrm{lfp}(p)$ and

$$\begin{array}{rl} \forall p' \in \mathit{ppoint} \quad p \xrightarrow{F^\sharp} p' \implies & F^\sharp(w_0(p)) \neq \top_s \\ & \wedge\ F^\sharp(w_0(p)) \sqsubseteq w_{old}(p') \\ \wedge\ p \xrightarrow{F^\sharp} \implies & F^\sharp(w_0(p)) \sqsubseteq_v \mathrm{t_{ret}} \end{array} .$$

  Therefore the set of $l^\sharp$ satisfying the condition of the algorithm contains $w_0(p)$ and there exists a maximal such $l^\sharp$ that is greater than $w_0(p)$. This proves that the execution does not get stuck at this step, and furthermore, as $w = w_{old}[p \leftarrow l^\sharp]$ and $l^\sharp \sqsupseteq w_0(p)$, we conclude that $w \sqsupseteq w_0$.

This proves the above invariant. By the first two points of the theorem, we conclude that the algorithm terminates by returning a witness w without conjunction. □

5.1.3 Java programs

In the Java language, all variables are declared with a fixed element of *type* (actually, the basic types are not exactly the same between Java and the Java bytecode: the smaller integer types are merged with int in the latter). This type must satisfy the same constraints that are expressed by the abstract semantics in the previous section (including the ones for interfaces) and can therefore be considered as a witness for each method, where the type of every variable is the same regardless of the program point. The difference is that the variables are the source variables, not the bytecode local variables and stack positions.

If the compiler does not transform the structure of the program too much, more precisely if each variable of the source program is mapped to a (bytecode) local variable in a given subset of the (bytecode) program points, without overlapping, then we see that the witness representing the typing of the source code can be renamed to a corresponding witness on the bytecode. This witness has an interesting feature: it does not contain any conjunction (because variables are declared with a single type). Therefore, for bytecode compiled from Java with a "natural" compiler, there exists a conjunction-free witness for every method (and thus the algorithm of the previous section will find it).

An alternative solution for introducing the verification of interfaces in a lightweight verification process in the case of Java (source) programs is to generate a stack map from the type annotations present in the source code, during the compilation process. Indeed, as the lightweight verification paradigm is being generalized to J2SE Java [JSR06], the task of generating stack maps is moving from a dedicated "preverifier" program to the compiler itself. One disadvantage of this technique is that all the tools that are used to manipulate bytecode (notably the compiler) must take care of stack maps consistently, which can be an overhead for their design. This is why we follow a different strategy, which extract a witness without conjunction directly from the bytecode (given that there exists one).

```
public void main(String  args) {
    I1 i1 = args.length > 0 ? new A() : new B();
    I2 i2 = args.length > 0 ? new A() : new B();
    i1.m1();
    i2.m2();
    Object i = args.length > 0 ? i1 : i2;
    i.toString();
}
```

Figure 7: A Java program for which a conjunction-free witness cannot be build from the atomic types of the least fix-point. `I1`, `I2`, `A` and `B` are defined as in the previous example (see Figure 5a-b).

5.2 Efficient fix-point pruning

Although the first algorithm is complete, it takes potentially a very long time, since the search space is the product over program points and local variables of the part of the type hierarchy that is greater than the corresponding value type in the least fix-point. In fact, it is rarely necessary, for a given variable and program point, to consider the entire type hierarchy above the type given by the least fix-point. Most of the time it is enough to choose one of its conjuncts (if it is a conjunction). The resulting pruning algorithm still has an exponential complexity, but it performs reasonably fast in practice. We exhibit an artificial Java program for which this doesn't work, but we could not find any counter-example in our (substantial) case studies, which indicates the applicability of the more efficient algorithm is very general.

5.2.1 Reducing the branching factor

In most cases, the following holds: there exists a witness w $\in$ *map* without conjunction such that the atomic types that appear in w are atoms of the corresponding conjunctions in the least fix point.

$$\forall p \in \mathit{ppoint} \quad \forall x \in \mathit{var} \quad \mathrm{w}(p)(x) = t \in \mathit{type} \implies \mathrm{lfp}(p)(x) = t \ \wedge \cdots$$

Counter-example This is not true for the program in Figure 7. In this example, we build two variables `i1` and `i2` with most precise type `I1` $\wedge$ `I2`. The variable `i` is then defined as the "union" of the two, and its type is therefore `I1` $\wedge$ `I2`. However, in a stack map without conjunction, the type of `i1` must be `I1`, and the type of `i2` must be `I2` (because we call `m1` and `m2`, respectively). Therefore, the type of `i` must be greater than the least upper bound of `I1` and `I2`, *i.e.*, `java.lang.Object`, which is not an atom of `I1` $\wedge$ `I2`.

5.2.2 Algorithm

Taking into account the hypothesis of section 5.2.1, we proceed by searching for a witness satisfying this hypothesis. Figure 8 describes the optimized algorithm. The only modification with respect to the first version is the set in which $l^\sharp$ is chosen.

Correctness The new algorithm is also sound and, though it is incomplete (see the above counter-example), it will succeeds in finding a witness if the hypothesis of section 5.2.1 holds.

let $w = \lambda p \in ppoint.\lambda x \in var.\top_v$ and $W = ppoint$
while $W \neq \emptyset$ do
take $p \in W$ (and remove it)
choose a maximal $l^\sharp \in state^\sharp$ such that
$\forall \boldsymbol{x \in var}$ $\quad \boldsymbol{l^\sharp(x) = \top}$
$\boldsymbol{\vee\ l^\sharp(x) = t \in type\ \wedge\ lfp(p)(x) = t\ \wedge \cdots}$
$\boldsymbol{\vee\ l^\sharp(x) = null = lfp(p)(x)}$
and $l^\sharp \sqsubseteq w(p)$
and $\quad \forall p' \in ppoint \quad p \xrightarrow{F^\sharp} p' \implies F^\sharp(l^\sharp) \neq \top_s \wedge\ F^\sharp(l^\sharp) \sqsubseteq w(p')$
$\wedge\ p \xrightarrow{F^\sharp} \implies F^\sharp(l^\sharp) \sqsubseteq_v \mathrm{t_{ret}}$
if $l^\sharp \neq w(p)$ then
$w := w[p \leftarrow l^\sharp]$
$W := W \cup \{p' \mid p' \xrightarrow{F^\sharp} p\}$
end if
done
return w

Figure 8: Efficient pruning algorithm. Changes with respect to the previous algorithm are shown in bold.

If not, the search will fail and the complete algorithm presented before should then be run instead.

Theorem 3 *The optimistic pruning algorithm always terminates, either by returning a stack map w or by a failure in the choice of $l^\sharp$. If it terminates by returning some w, then w is a witness without conjunction. Furthermore, if a witness w without conjunction exists whose atoms are in* lfp*, the algorithm may return w.*

The proof is identical to the proof for the complete algorithm.

6 Experiments

We have implemented the ideas presented here in a verifier for real Java bytecode. We discuss the differences in the concrete implementation and give some experimental results.

6.1 Implementation

A prototype analyser/verifier was developed in Ocaml, using Javalib. It reads Java class files and adds the stack maps as method attributes, as defined in the Java Virtual Machine specification.

6.1.1 Extensions

The bytecode language presented so far is considerably simplified. We list the differences with the real bytecode that we had to address.

- First, the Java bytecode uses an operand stack in addition to local variables. This complicates the abstract states, as they now have another list of abstract values, of variable length. Of course, this adds more reasons for the verification to fail, namely, (operand) stack overflow or underflow, or the possibility to have different stack heights at some program point.

- In addition to 32-bit integers, Java bytecode has floats, longs and doubles. floats are easily abstracted by $\top_v$, and the 64-bit types by two $\top_v$s. Note that, although 32-bit integers are not distinguished from shorter integers at the bytecode level (they are in the source code), this is not the case for arrays of such types. Therefore, to ensure that the array bounds checks performed at runtime correctly interpret the length field, the size of elements must be known. This implies that arrays of floats or ints must not be confused with arrays of shorts. However, individual float and int values can still be merged, since the instruction for accessing arrays are typed.

- From the verification point of view, exceptions just add some more transitions in the control flow graph, with a semantics that empties the stack and then push some constant reference type (the type that is caught by the corresponding handler). They pose no particular difficulty.

6.1.2 Limitations

While analysing real bytecode, we have omitted some aspects of the verification even in the concrete implementation:

- Subroutines complicate the verification, and are not generated anymore by compilers. We did not address this issue and the prototype only applies to code without sub-routines. The adaptation of abstract interpretation to sub-routines was studied in [Qia00], and a lightweight verifier capable of handling sub-routines is presented in [Kle03].

- In addition to memory safety, the original type system used by the bytecode verifier ensures that any object is initialized before it is "used". This property is crucial to security in some cases. For example, to ensure some high-level access property, an API may rely on the fact that, because of some classes being final, some type of objects can only be created by a known set of trusted constructors (at the source level) that ensure a common invariant, but obviously this only holds if some constructor (some initializer at the bytecode level) is invoked. Similarly, bytecode verification checks that only one constructor invocation occurs on each object for every super-class of this object, in the appropriate order.

  We did not implement those verifications, because this is a distinct concern (the memory safety property can be stated without it) whose treatment amount to adding some more types (for uninitialized objects) to the abstract domain, which poses no particular difficulty (alternatively, this can also be done as another verification pass).

Also, note that the semantics that we gave to the bytecode used big-step calls, which prevents us to consider even a simplified (interleaving) version of concurrency, which would require explicit call stacks. Therefore, in principle, all the results presented here only applies to single-threaded programs. However, the scheme of the proof (an invariant that holds at each state of a small-step semantics) does not seem to rely on sequentiality, and we believe that there is no issue in extending it to threads.

6.1.3 Stack maps and Checking

The stack maps that are attached to th bytecode are not exactly a representation of conjunction-free witnesses, but only of the value of such witnesses for a subset of the program points. These points correspond basically to the basic blocks of the control flow graph. This reduces the size of stack maps, while still allowing a very simple checking algorithm that evaluates program points in order. We will not detail this aspect, as we used the same subset of program points and the same checking algorithm as Sun's lightweight bytecode verifier.

The resulting stack maps are encoded in the class files either as StackMap attributes in the same format as the lightweight bytecode verifier, or with a new attribute using a sparse representation. In the latter, we just replaced an array of value types by an array of bits (to indicate which values are not $\top_v$) followed by the list of non-$\top_v$ values. In order for the comparison to be fair, the sparse representation uses the same verbose encoding of value types as the stack map representation.

6.2 Results

The following benchmarks were used to experiment with the analysis, pruning and lightweight checking with interfaces.

6.2.1 Case studies

We have successfully performed the lightweight verification with interfaces of the following programs.

- 433 old midlets (Java applets for mobile phones) from midlet.org.
- Soot 2.2.4: a framework for analysing Java bytecode
- Eclipse SDK 3.2.2 with all included plug-ins

All methods have been checked, up to the following limitations:

- The (rare) methods that contain sub-routines have not been analysed.
- Some methods that referred to unavailable libraries have not been analysed.
- Dead code in some methods lead to least fix-points with $\bot_s$ for some program points. The pruning algorithm does not apply in this case.
- In Eclipse, some packages seem to co-exist in different versions that use the same class names with different hierarchies. As we built conservatively the complete hierarchy of the distributed classes, some methods in one version could not type-check under the union of both hierarchies[5].

In all cases, a stack map without conjunction could be obtained from the atoms of the least fix-point, thanks to the optimistic heuristic presented in Section 5.2.

6.2.2 Computing time

Figure 9 shows the main interesting computing times for the two case studies. The first

[5]Some classes even exist with different super-classes. In this case we just choose one, which is definitely not safe.

	jar size	analysis + pruning	analysis	pruning	checking
midlets	11M	5m54	23%	77%	0m23
Soot	4.4M	3m40	17%	83%	0m11
Eclipse	96M	24m43	26%	74%	1m55

Figure 9: Computing time

column gives the size of the benchmarks (jar files). The second one shows the total time taken by the complete stack map generation procedure (on the productor's side). This time is then divided into the analysis phase (third column) and the pruning phase (fourth column). The last column correspond to the checking time (consumer's side). Clearly, most of the time is spent in pruning, but even this time remains acceptable (three to six times the cost of the analysis), especially since this operation only needs to be performed once, by the code producer. The checking time is short and could be further reduced with a reasonably optimised implementation.

6.2.3 Increasing the proportion of $\top_v$

Figure 10 estimates the size of witnesses before and after pruning, in terms of the proportion

proportion of non-$\top_v$ in	lfp	pruned stack map	ratio
midlets	44%	34%	77%
Soot	67%	42%	63%
Eclipse	58%	39%	66%

Figure 10: Number of non-$\top_v$ values

of pairs p, x for which the value is not $\top_v$. The last column shows the proportion of "positions" (of pairs p, x) that are kept with a non-$\top_v$ value by pruning. We see that the "initial" proportion of non-$\top_v$ is greater in Soot and Eclipse, which indicates that objects (or arrays) are used more often in Eclipse than in midlets (remember that base types are abstracted by $\top_v$). Also, the pruning removes more values in Soot and Eclipse than in midlets (which is not surprising since there are more non-$\top_v$ values to remove, in proportion). In the end, the numbers of non-$\top_v$ in the stack maps are very close for the three test cases.

6.2.4 Certificate size

The first four sub-columns of Figure 11 give the space saved by pruning, both for the class

witness	fix-point		pruned witness			
representation	extensive		extensive		sparse	
format	.class	.jar	.class	.jar	.class	.jar
midlets	19.8%	7.3%	17.4%	6.8%	16.0%	7.0%
Soot	14.0%	7.2%	11.5%	6.5%	11.5%	7.3%
Eclipse	11.8%	3.3%	9.5%	3.0%	9.6%	3.2%

Figure 11: Size reduction by pruning and sparse representation

files and the jar files (compressed archives). The numbers correspond to the difference in size with respect to the same file format without stack map. For example, the total jar size for Eclipse with pruned stack maps included is 3.0% greater than the original jar files (without stack maps). In the two columns for the fix-point, since only conjunction-free witnesses can be encoded in class files, we did not include any stack map for the methods whose least fix-point had conjunctions (which is actually quite rare). Note that we can only underestimate the benefit of pruning by doing this. We see that in the case of midlets, for example, the size of the stack maps is reduced from 19.8% to 17.4% of the total initial class files, or from 7.3% to 6.8% of the initial jar files. Therefore there is no significant improvement here since the size of what is shipped (*i.e.*, the jar files with stack maps) is only reduced by less that one percent.

The last two sub-columns of the figure show the effect of a sparse representation of the stack maps obtained after pruning. We see that a sparse representation has little impact on the size, and that the small savings that we get for (some) class files are canceled by the compression phase, and tend to yeld larger jar files (even if the eight-bit alignment of the class files is kept).

We have not tried to encode our stack maps with the new StackMapTable attribute defined by JSR202, which is more complex and was designed to factorize most of the information. The results would probably be quite different since this format relies on the assumption that the type of variables do not change too often, while the pruning may for example set any variable to $\top_v$ even if it was not modified, as soon as the type information for this variable is not needed anymore.

7 Related work

The formalisation of Java bytecode verification has received a lot of attention. We can only point to a few relevant papers.

Java bytecode verification Freund and Mitchell [FM03] prove the soundness of a type system for a very large subset of the Java bytecode with respect to a small-step operational semantics (with explicit stacks). Their model of states is close to ours, but instrumented by tags that keep track of the type of every value. They do not address the problem of inferring types in presence of interfaces. A survey of bytecode verification techniques and solutions to various known difficulties (interfaces, object initialisation, sub-routines) can be found in [Ler03]. The concept of lightweight verification, which is now used in J2ME, was introduced by Rose [Ros03]. Several algorithms were given, with enhancements that allow to reduce the number of program points for which a stack map is necessary, more than what is done in Sun's lightweight bytecode verifier. The issue of verifying interfaces was not considered.

Using sets of types to verify interfaces Knoblock and Rehof [KR01] analyse an SSA form of the Java bytecode in the Dedekind-MacNeille completion of the type hierarchy, and they show that this minimal completion achieves an optimal precision, *i.e.*, every program typable in the power set completion is typable in the Dedekind-MacNeille completion. The analysis presented in section 3 is therefore very similar to their work. Our representation of the domain differ, though: we use conjunctions of types rather that disjunctions (in both cases, upward/downward-closed sets are not represented in extension). The lattice that we use to abstract values is close to the ideal completion of the type hierarchy (it is a super-set of the ideal completion because the latter further requires that conjunctions be "not empty", in the sense that they must have a lower bound in the hierarchy). Furthermore,

our analysis only uses the subset of $value^\sharp$ that is obtained by taking upper bounds of atomic types, which is isomorphic to the Dedekind-MacNeille completion of $type \setminus \{\text{int}\}$. See [DP90] for an account on completion techniques for posets. Knoblock and Rehof do not prove the correctness of their analysis with respect to a concrete semantics and safety property. Qian [Qia99] proposes a type system for Java bytecode that uses arbitrary disjunctions of reference types to allow the static verification of interfaces. Several safety properties are proved for typable programs (type preservation, possible uses of uninitialized objects, of sub-routines return addresses). The actual inference of types is not detailed. Push [Pus99] has formalised a variant of Qian's bytecode verifier in HOL and proved its correctness with respect to a small-step operational semantics. Again, concrete values are tagged with their type. Goldberg [Gol98] focuses on dynamic loading of classes and proposes a framework for verifying Java class files out of order, while ensuring the global soundness of typing. Class files are verified by a data-flow analysis that uses disjunctions of types (which solves the problem of not knowing the type hierarchy) and yields the minimal set of ordering constraints between types under which the class is type-safe. These constraints are added to a global typing context that is transmitted across invocations of the verifier, and the global safety is defined as the consistency of this context.

Pruning We have previously proposed a pruning algorithm for getting weaker abstract interpretation witnesses [BJT07]. Such pruning algorithms were independently studied by Seo, Yang, Yi and Han [SYYH07]. The problem that we consider here is different: the goal is not to get a maximal witness in a given lattice, but to get a witness without conjunction, a property that is not monotone. Therefore, directly applying one of the algorithms from [BJT07] would not necessarily help in getting such a witness. The backward computation that we proposed in the same work for distributive analyses (which is the case of the bytecode verification) does not apply either, as shown in the introduction: the backward algorithm performs greatest lower bound operations, which in the present setting introduce conjunctions.

8 Conclusion

We have shown how the notion of pruning provides a viable means of integrating the verification of interfaces into lightweight bytecode verification. This is achieved by combining an extended bytecode analyser and an algorithm for removing conjunctions from the result of the analysis which, together, allows to compute stack maps where interfaces are treated on a par with other types.

The bytecode analysis that we have proposed here adds sets of types to the abstract domain in order to verify interfaces. The ensuing pruning step optimises the typing found by the analyser, reducing all such sets to a singleton, and removing as many typing information as possible while still ensuring the memory safety, *i.e.*, that all memory accesses will be to existing fields of objects. The resulting stack maps can be checked without any overhead compared to exisiting lightweight bytecode verification and will ensure statically the safety of interface method calls. We also show that it is possible to simplify several aspects of the BCV when constructing an abstract domain that is specific to the memory safety property. In particular, there is no need to distinguish between base types and it is even possible to identify these base type with the $\top_v$ element of the domain (which allows a program to use an address as an integer).

In terms of semantic correctness, we have shown that it is possible to reason directly with an untyped concrete semantics rather than a defensive virtual machine. Both techniques

are equally sound, but the latter requires an additional abstraction step that explains the link between the raw state model that we use and the tagged memory objects used inthe instrumented semantics. In other words, we use a notion of state that is closer to the actual implementation and, hence, more convincing. We made the reasoning modular by separating the interprocedural aspect of the semantics and soundness proof. This allows us to cut the double induction that arises from using a small-step semantics with big-step calls. In order to complete the picture, the semantics with big-step calls that we used should be related to a small-step semantics with a call stack, but we leave this for further work.

In terms of experiments, we have shown that the technique works well in practice, as we could successfully analyse a large set of Java class files. Furthermore, the idea is not relevant just for Java, but should apply to other object oriented languages with multiple inheritance, since it only relies on the transformation of the poset representing a type hierarchy into a lattice. The results show that it is feasible to compute efficiently conjunction-free stack maps with interfaces ; however, they are disappointing in terms of reducing the stack map size: even though a significant number of variables are set to $\top_v$ by pruning, this is not enough for a sparse coding to be more efficient than a naive coding of stack maps, especially as class files are eventually compressed.

As we said before, in this study we considered one aspect of the security of Java bytecode, *viz.*, the memory safety. Further work should extend the formalisation proposed here to prove that for example access control properties are also ensured by the verifier. In another direction, our stack map generator should be extended to produce stack maps in the StacMapTable format proposed for Java.

References

[BDJdS02] Gilles Barthe, Guillaume Dufay, Line Jakubiec, and Simão Melo de Sousa. A formal correspondence between offensive and defensive javacard virtual machines. In *VMCAI '02: Revised Papers from the Third International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, 2002.

[BJT07] Frédéric Besson, Thomas Jensen, and Tiphaine Turpin. Small witnesses for abstract interpretation-based proofs. In *ESOP '07: Proceedings of the 16th European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*. Springer, 2007.

[BLTY03] Gilas Bracha, Tim Lindholm, Wei Tao, and Frank Yellin. *CLDC Byte Code Typechecker Specification*. Sun Microsystems, 2003.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symp. on Principles of Programming Languages*. ACM Press, 1977.

[DP90] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 1990.

[FM03] Stephen N. Freund and John C. Mitchell. A type system for the java bytecode language and verifier. *Journal of Automated Reasoning*, 30(3-4), 2003.

[Gol98] Allen Goldberg. A specification of java loading and bytecode verification. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*. ACM Press, 1998.

[JSR06] JSR 202 Expert Group. *Java Class File Specification Update*, 2006. Sun Microsystems.

[Kle03] Gerwin Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[KR01] Todd B. Knoblock and Jakob Rehof. Type elaboration and subtype completion for java bytecode. In *ACM Transations on Programming Languages and Systems*, volume 23(2), 2001.

[Ler03] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4), 2003.

[LR98] Xavier Leroy and François Rouaix. Security properties of typed applets. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. ACM Press, 1998.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (2nd edition)*. Prentice Hall, 1999.

[Nec97] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*. ACM Press, 1997.

[Pus99] Cornelia Pusch. Proving the soundness of a java bytecode verifier specification in isabelle/hol. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Springer-Verlag, 1999.

[Qia99] Zhenyu Qian. A formal specification of java virtual machine instructions for objects, methods and subrountines. In *Formal Syntax and Semantics of Java*. Springer-Verlag, 1999.

[Qia00] Zhenyu Qian. Standard fixpoint iteration for java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4), 2000.

[Ros03] Eva Rose. Lightweight bytecode verification. *Journal of Automated Reasoning*, 31(3-4), 2003.

[SYYH07] Sunae Seo, Hongseok Yang, Kwangkeun Yi, and Taisook Han. Goal-directed weakening of abstract interpretation results. *ACM Transactions on Programming Languages and Systems*, 29(6), 2007.

A Summary of definitions

Object oriented structure

C, I, m, f	$\in$	*ident*			Java identifiers
C	$\in$	*class*	$\subset$	*ident*	class names
I	$\in$	*interface*	$\subset$	*ident*	interface names
t	$\in$	*type*	::=	int $\mid$ C $\mid$ I $\mid$ $t[]$	types
		super	:	$\mathit{class} \setminus \{\text{Object}\} \to \mathit{class}$	super-class
		implements	:	$\mathit{class} \to \mathcal{P}(\mathit{interface})$	directly implemented interfaces
		extends	:	$\mathit{interface} \to \mathcal{P}(\mathit{interface})$	direct super-interfaces
ms	$\in$	*msig*	$\subset$	$\{t.m(t_1, \ldots, t_n) \mid t \neq \text{int}\}$	method signatures
		arity	:	$\mathit{msig} \to \mathbb{N}$	arity
		result	:	$\mathit{msig} \to \mathit{type}$	return type
fs	$\in$	*fsig*	$\subset$	$\{C.f : t\}$	field signatures

Attributes of the current method

x, y, z	$\in$	*var*			local variables
		arg	$\subseteq$	*var*	formal parameters
		$\mathrm{t_{arg}}$	:	$\mathit{arg} \to \mathit{type}$	type of parameters
		$\mathrm{t_{ret}}$	$\in$	*type*	return type
p	$\in$	*ppoint*	=	$[0, \lvert \mathit{ppoint} \rvert - 1]$	program points
e	$\in$	*expr*	::=	$n \quad n \in [-2^{31}, 2^{31} - 1] \mid \text{null}$	expressions
			$\mid$	$y + z \mid \mathsf{new}\ C \mid y.fs \mid \mathsf{new}\ t[y]$	
			$\mid$	$y[z] \mid y.ms(x_1, \ldots, x_{\mathrm{arity}(ms)})$	
i	$\in$	*instr*	::=	$x := e \mid x.fs := y \mid x[y] := z$	instructions
			$\mid$	$\mathsf{goto}\ p \mid \mathsf{if}\ x < y \quad p \mid \mathsf{return}\ x$	
		code	:	$\mathit{ppoint} \to \mathit{instr}$	method code

Figure 12: Idealized Java bytecode

values and states

v	$\in$	*word*			32-bit values
		fields	:	$\mathit{class} \rightarrow \mathcal{P}(\mathit{fsig})$	transitive fields of a class
		object	::=	$\langle C, o\rangle \quad o : \text{fields}(C) \rightarrow \mathit{word}$	objects
		array	::=	$\langle t, a\rangle \quad a : [0, n-1] \rightarrow \mathit{word}, n \geq 0$	arrays
h	$\in$	*heap*	=	$\mathit{word} \setminus \{0\} \rightharpoonup (\mathit{object} \cup \mathit{array})$	heaps
		alloc	:	$\mathit{heap} \rightharpoonup \mathit{word} \setminus \{0\}$	memory allocator
l	$\in$			$\mathit{var} \rightarrow \mathit{word}$	value of local variables
s	$\in$	*state*	::=	$\langle h, l, p\rangle$	program states

typing

	$\preceq$	$\subseteq$	$\mathit{type} \times \mathit{type}$	subtyping order
	$\cdot \vdash \cdot : \cdot$	$\subseteq$	$\mathit{heap} \times \mathit{word} \times \mathit{type}$	dynamic typing

transition relations

$\rightarrow$	$\in$	*bigstep*	=	$\mathcal{P}\left(\bigcup_{n\geq 0} \begin{array}{c}(\mathit{heap} \times \mathit{word} \times \mathit{word}^n) \\ \times(\mathit{heap} \times \mathit{word} \cup \{\text{error}\})\end{array}\right)$	big-step relations
		$\dot{\rightarrow}$	:	$\mathit{msig} \rightarrow \mathit{bigstep}$	semantics of method calls
		$\rightarrow$	$\subseteq$	$\mathit{state} \times (\mathit{state} \cup \mathit{heap} \times \mathit{word} \cup \{\text{error}\})$	small-step semantics of the current method

Figure 13: Semantic definitions

domain

$v^\sharp$	$\in$	$value^\sharp$	$::=$	null $\mid$ $\top_v$	abstract values
			$\mid$	$t_1 \wedge \cdots \wedge t_n \quad n \geq 1, t_i \in type \setminus \{\text{int}\},$ $\forall i,j \leq n \quad t_i \preceq t_j \implies i = j$	
$s^\sharp$	$\in$	$state^\sharp$	$=$	$var \to value^\sharp$	abstract states
m	$\in$	map	$=$	$ppoint \to state^\sharp$	stack maps
$s^\sharp$	$\in$	${state^\sharp}_{\perp_s}^{\top_s}$	$=$	$state^\sharp \cup \{\perp_s, \top_s\}$	

concretisation functions

$\gamma_h, h \in heap$	:	$value^\sharp \to \mathcal{P}(word)$	for values
$\gamma_p, p \in ppoint$	:	${state^\sharp}_{\perp_s}^{\top_s} \to \mathcal{P}(state \cup \{\text{error}\})$	for states
γ	:	$(value^\sharp \cup \{\top_s\})$ $\to \mathcal{P}((heap \times word) \cup \{\text{error}\})$	for return states

partial order

$\sqsubseteq_v$	$\subseteq$	$value^\sharp \times value^\sharp$	for values
$\sqsubseteq$	$\subseteq$	${state^\sharp}_{\perp_s}^{\top_s} \times {state^\sharp}_{\perp_s}^{\top_s}$	for states

least upper bound

$\sqcup_v$	:	$value^\sharp \times value^\sharp \to value^\sharp$	for values
$\sqcup$	:	${state^\sharp}_{\perp_s}^{\top_s} \times {state^\sharp}_{\perp_s}^{\top_s} \to {state^\sharp}_{\perp_s}^{\top_s}$	for states

abstract transition relations

$\longrightarrow$	$\subseteq$	$ppoint \times (state^\sharp \to {state^\sharp}_{\perp_s}^{\top_s}) \times ppoint$	
$\longrightarrow$	$\subseteq$	$ppoint \times (state^\sharp \to value^\sharp)$	method return

Figure 14: Abstract semantics

body	:	$msig \rightharpoonup \{(var, arg, ppoint, \text{code})\}$	code
$\underbrace{\dot{\rightarrow}, (var, arg, ppoint, \text{code})}_{\longrightarrow}$		$\in bigstep$	method execution $w.r.t.\ \dot{\rightarrow}$
lookup	:	$(type \setminus \{\text{int}\}) \times msig \rightharpoonup class$	resolution
$\underbrace{\dot{\rightarrow}, ms}_{\longrightarrow}$	$\in$	$bigstep$	signature invocation $w.r.t.\ \dot{\rightarrow}$
S	:	$(msig \rightarrow bigstep) \rightarrow (msig \rightarrow bigstep)$	
	$=$	$\lambda \dot{\rightarrow} .\lambda ms. \underbrace{\dot{\rightarrow}, ms}_{\longrightarrow}$	
$\dot{\longrightarrow}$	:	$msig \rightarrow bigstep$	interprocedural
	$=$	lfp(S)	semantics
	$=$	$\lambda \dot{\rightarrow} . \bigcup_{d \geq 0} \left((\mathrm{S}^d (\lambda ms \in msig.\emptyset)) \, (ms) \right)$	

Figure 15: Interprocedural layer. The variable $\dot{\rightarrow}$ represents a function mapping signatures to bigstep transition relations, *i.e.*, $\dot{\rightarrow} : msig \rightarrow bigstep$.